Exercise 4: AVR32 FreeRTOS

Last week you used the AVR32 without an operating system. This week we will use a very small operating system called FreeRTOS. It is a small and simple operating system kernel, with features you expect from a real-time system. Many embedded microcontrollers are supported, including AVR32 UC3. Last year we used 8-bit AVR with FreeRTOS in this course, and even if that is a very low powered device, it still worked well.

Description of how FreeRTOS works, how to use it and an API can be found at http://www.freertos.org/.

The purpose of this exercise is NOT necessarily to get the best results on the tests with the BRTT. Don't worry if your results will be worse than when you used interrupts last week, it will be very hard to beat those results, and not possible with FreeRTOS.

1. Run Demonstration Program

Open the FreeRTOS AVR Studio 5 project from it's learning. The default program does exactly the same as the default program from last week, it blinks a LED and prints "tick" to the serial console. The difference is that this is performed in a FreeRTOS task, which will be described below. You should compile and run the program, and verify that the LED and printing works as expected.

Notice that the compile time is longer. This is due to the additional FreeRTOS code.

For an unknown reason, `printf()` is not working normally with FreeRTOS. Printing simple strings should be OK, but using `printf()` to print the valuables of variables will most prevent the whole program to misbehave.

2. Creating FreeRTOS tasks

When programming AVR32 directly like we did last week there is really no concept of concurrency. There is only one process, the one running the `main()` function. The only way to execute code other than what is defined in the `main()` function is by using interrupts.

With FreeRTOS it is possible to create tasks, and FreeRTOS will schedule which of these tasks that will run at any given time. Below is a list for functions you need for creating tasks and run them in a real time environment. For details about the functions and their parameters you need to look at the API on the FreeRTOS webpage (http://www.freertos.org/a00106.html), or other documents you might find online.

- xTaskCreate() – Creates tasks
- vTaskStartScheduler() – Start real-time scheduling, must be called after task have been created

Remember you also need to create the function the task is supposed to execute.

When creating a task the stack size can be set to `configMINIMAL_STACK_SIZE`, and priority should be set to a value higher than `tskIDLE_PRIORITY`.

Assignment A:

Create two tasks from the following pseudo code:

```
TaskA()
{
      While (true)
      {
            Toggle LED 0;
            delay_ms(200);
      }
}

TaskB()
{
      While (true)
      {
            Toggle LED 1;
            delay_ms(500);
      }
}
```

You should use the FreeRTOS function vTaskDelay() for delay, not the busy-delay from last week. This is already used in the demonstration project. Observe that the two LEDs will blink with different frequencies.

3. Reaction test

We will now run the same test as we did last with, just using FreeRTOS instead of "normal" AVR32 programming. Since we are now able to create tasks, we can create one task for each of the three signals we need to listen to. Each of these tasks should listen and set its response signals as soon as it receives the test signal from the BRTT.

It is probably possible to get better results on the test by only using one task to answer all signals. You are still asked to use three tasks to learn how to use the FreeRTOS system and other similar systems.

If three tasks are created with the same priority, FreeRTOS will schedule between them using round robin. The scheduling is preemptive, and will interrupt a task while it is running if a higher priority task wants to run, or it is time to run another task of the same priority.

The FreeRTOSConfig.h file defines the CPU frequency of the AVR and the tick frequency. The CPU frequency of the AVR32 UC3-A3 is 12MHz. The tick is used by the FreeRTOS to measure time, and the tick frequency decides how often a tick occurs. The default is 1000 Hz, which means that the shortest delay or period possible is 1 ms. There is no need to change the tick frequency in this exercise.

Assignment B:

Create three tasks, one for test A, B and C. The tasks should detect a test signal from the BRRT by continuously reading the value (busy-wait). When the signal is detected, the task should send the response signal by pulling its voltage low. You need to keep the value low for a while so the BRTT can detect the signal (vTaskDelay).

Do an A+B+C reaction test with 1000 subtests and store the results.

Try to give the on of the tasks a higher priority than the rest, what happens and why?

3.1. CPU Usage

Your program from assignment B will utilize all the CPU cycles that are available to check the value on either of the input signals. This ensures a fast response time at the cost of terrible waste of CPU cycles. To demonstrate this, you should add a 4th task to your program that runs the following code:

```
static void vCpuWork(void *pvParameters)

{

      int i;

      double dummy;


      while(1)

      {
```

```
            for(i=0; i<1000000; i++)
            {
                  dummy = 2.5 * i; //not important
            }
            gpio_toggle_pin(LED0_GPIO);
      }
}
```

Run first this task alone, and observe that it will blink the LED about once a second. Between each time the LED toggles, a large number of CPU cycles are spent doing calculations. This means that the less of the CPUs time that is given to this task, the slower the LED will blink.

Do the same test as in assignment B, with the addition of this 4th task running with lower priority than the other three tasks. This simulates a low priority, CPU intensive task that runs in the background while the other tasks respond to the test. Observe and explain what happens.

4. Reaction test using periodic polling

In assignment B, each task used busy-wait to detect signals. This is, as mentioned earlier not the most elegant and not the most effective way to do something. A high priority task using busy wait will prevent lower priority tasks from running, as you should have seen when introducing the 4th task. A better solution is to read the input, put the task to sleep for a while before reading the input again. This is often done periodically.

In FreeRTOS we can create accurate periodic tasks using the `vTaskDelay()` function or the `vTaskDelayUntil()` function. Find out what the difference between these two are, and understand why `vTaskDelayUntil()` is the best for creating accurate periodic tasks.

The `portTICK_RATE_MS` *constant tells you how many ms one tick is.*

Assignment C:

Make the three tasks from B periodic, with a period of 5 ms.

Do an A+B+C reaction test with 1000 subtests and compare with the results from B (the response will be significantly slower).

As in B, try to give one of the tasks a higher priority than the rest, what happens and why? You can try delaying with a busy wait when lowering the pin.

You can also experiment with other periods, but be aware that the minimal period length is 1 ms, since this is the length of the tick used by the FreeRTOS scheduler.

Run the same test together with the 4th task with lower priority, and explain what happens.

5. Approving the exercise

For this exercise you need to show the following to a student assistant to get the exercise approved.

1. Run the reaction test using periodic polling, which the 4th task running at low priority, you don't need to show the busy-wait test since it is considered an easier exercise.
    - Explain how the 4th task is affected by the two methods.
2. Compare the results from busy wait and periodic polling.
    - Which one performed best, why?
    - How did they react to running one of the tasks as a higher priority?

6. Appendix: Tips and Hints

Problems reprogramming the UC3:

If the application doesn't start after programming the device, try to reboot the device by disconnecting and connecting the usb-cable. For some reason the UC3 does not restart properly when reprogrammed.

Finding source implementations:

Try marking a function and press alt-g. This will take you to the implementation of that function.