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.

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

Remember that the desired properties of a real-time system is consistent and predictable timings, not necessarily fast ones. It will be very hard to beat the fast timings from interrupts last week, and not possible at all with FreeRTOS. When comparing results to Exercise 3, look at the shape of the distributions and the relative timings between the response tasks, not the absolute timings.

1. Run the demo program

Download and open the FreeRTOS Atmel Studio project from Blackboard. The default program does exactly the same as the default program from last week, it blinks an 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.

`printf()` *can sometimes use a large amount of memory, depending on what you are printing out. Printing a simple string uses very little, but formatting just a single integer won't even run unless the stack size is at least 1024 bytes. Since this microcontroller only has 64KB memory, it means you can't start a whole lot of threads that will use printf. It likely won't be a problem today, just be aware that a) you can increase the stack size if printf is misbehaving, and b) you can't keep doing this forever.*

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 one 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 tasks 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 by the third parameter to xTaskCreate, and should be no lower than configMINIMAL_STACK_SIZE (default 256). The priority should be set to a value higher than tskIDLE_PRIORITY, and no larger than configMAX_PRIORITIES (default 8).

Task A:

Create two tasks that periodically toggle two LEDs. The first task should toggle LED0 every 200ms, and the second task should toggle LED1 every 500ms. You should use `vTaskDelay()`, not the busy delay.

Make sure the LEDs blink at the correct frequencies by measuring with a stopwatch. It doesn't have to be completely accurate, we're just checking that everything works as expected.

3. Reaction test

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

FreeRTOS uses round-robin scheduling between tasks of equal priority. The scheduling is preemptive, which means it can interrupt a task while it is running at any point, where it will swap to a higher priority task or a different task of the same priority. This occurs whenever a scheduler timer interrupt occurs (a "tick"), or the current task triggers a reschedule via a call to functions like yield, delay, or functions associated with things like queues and semaphores.

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 1000Hz, which means that the shortest delay or period possible is 1ms. This is also the fastest tick rate we can run on this platform, due to integer math limitations in FreeRTOS.

In the next few tasks we will be recording results from 5 different tests. It is recommended that these results in the same spreadsheet to make it easier to compare them. It is also recommended that you parameterize the contents of the tasks (either with the task arguments or with #define's) wherever appropriate, to avoid any copy-paste-edit errors. Here's one possible method:

```
struct responseTaskArgs {
    struct {
        uint32_t test;
        uint32_t response;
    } pin;
    // other args ...
};

static void responseTask(void* args){
    struct responseTaskArgs a = *(struct responseTaskArgs*)args;
    while(1){
        if(gpio_pin_is_low(a.pin.test)){
            // ...
        }
    }
}

int main(){
    ...
    xTaskCreate(responseTask, "", 1024,
        (&(struct responseTaskArgs){{TEST_A, RESPONSE_A}, other args...}),
        tskIDLE_PRIORITY + 1, NULL);
}
```

Task B:

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

Perform two tests, both with 1000 iterations. For the first test use `vTaskDelay(1)` to keep the value on the pin low until the next tick, and for the second test use `busy_wait_short()` (which is approximately 5us).

Calling `vTaskDelay()` *forces the scheduler to reevaluate what task to run. A value of 0 will put the calling task at the back of list of runnable tasks (which means it will run again immediately if there are no other runnable tasks), a value of 1 will make this task runnable when the next scheduler timer tick occurs, and a value greater than this will delay the wake time further.*

Compare the worst-case response times, and explain why the distributions look different.

Task C:

As in exercise 3, we will give one of the tasks some additional work to do before it responds to its signal. Again, the "work" is just a call to `busy_wait_ms()`.

Make task C wait 3ms before responding, while tasks A and B respond immediately. As before, all tasks should busy-wait on the input pin. Use `vTaskDelay(1)` to keep the response pin low until the next tick.

Ideally, task C should respond quite soon after 3ms has passed. Why doesn't it?

Optional: Same as in Task B – compare `vTaskDelay(1)` and `busy_wait_short()`. Why does changing to `busy_wait_short()` impact the C response time so severely, compared to A and B? (This is really the same question as in Task B)

Why would changing the priorities of the tasks not help us improve the responsiveness of the C task? Hint: try it and see what happens.

Task D:

The setup for this question contains spoilers to the answer for the question above. If you don't want spoilers, this is where you stop reading for now. You have now continued reading, so here's the spoiler: By busy-waiting on the test pin, we continue using CPU resources even though we have just completed our own response (which means the pin will remain high until the BRTT starts a new test), thus blocking other tasks from running until the scheduler tick occurs. By changing the priorities, the low priority task is completely blocked from checking its test pin, since any higher priority tasks always have "work" to do.

So instead of busy-waiting for the test pin to go low, we only check it intermittently. If the pin is *not* low, we wait for the next scheduler tick by calling `vTaskDelay(1)`, then loop around and test the pin again once the scheduler wakes us. Note that this is *not* periodic execution – if the pin is low we perform some amount of work (`busy_delay_ms`) that takes longer than the "period", which means that we will overrun our target period in the case where the test pin is low and we are responding to it.

The core point is that *all tasks must yield*, otherwise the highest priority task will never let the lower priority tasks run at all. And now that we yield *outside* the response operation, we can use either `busy_wait_short()` or `vTaskDelay(1)` to keep the response pin low.

Why doesn't the way we keep the response pin low impact the response times any more?

Make all tasks call `vTaskDelay(1)` in the loop that reads the test pin, instead of busy-waiting. As before, A and B respond immediately, while C has to perform 3ms of work first. Record the response time results, and compare to the busy-waiting method.

What happened to the response times of each of the tasks? Why?

Task E:

Use priorities to make the response times for tasks A and B better.

What are the worst-case response times for A and B, and why?

Task F (optional):

Use `vTaskDelayUntil()` to perform periodic polling of the test pin. Make sure the period is long enough to accommodate the time it takes to respond to all the test pins.