Lab 5 Real-Time Scheduling

Deadlines & Grading

- **Code and report submission**: Friday, April 26 at 9:00 PM on CMS, **groups of 2**
- **Grading**: 7% of total grade

Section I: Overview

Goal. The purpose of this lab is to implement real-time scheduling. To successfully complete this lab, you will need to understand the following:
- Concurrency (Lab 3)
- Real-time scheduling

NOTE 1 We suggest that you first understand real-time scheduling on paper before you start writing C code. It is important that you have confidence in your implementation before you start, because it is challenging to find mistakes using standard testing methods.

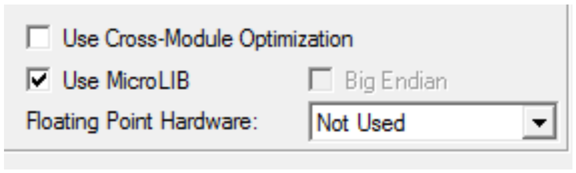
Precautions.
- The micro-controller boards should be handled with care. Misuse such as incorrectly connecting the boards is likely to damage the device.
- These devices are *static sensitive*. This means that you can "zap" them with static electricity (a bigger problem in the winter months). Be very careful about handling boards that are not in their package. Your body should be at the same potential as the boards to avoid damaging them. For more information, check wikipedia.
- It is your responsibility to ensure that the boards are returned in the same condition as you received them. If you damage the boards, it is your responsibility to get a replacement ($35).

Section II: Getting Started

Part 1: µVision Setup

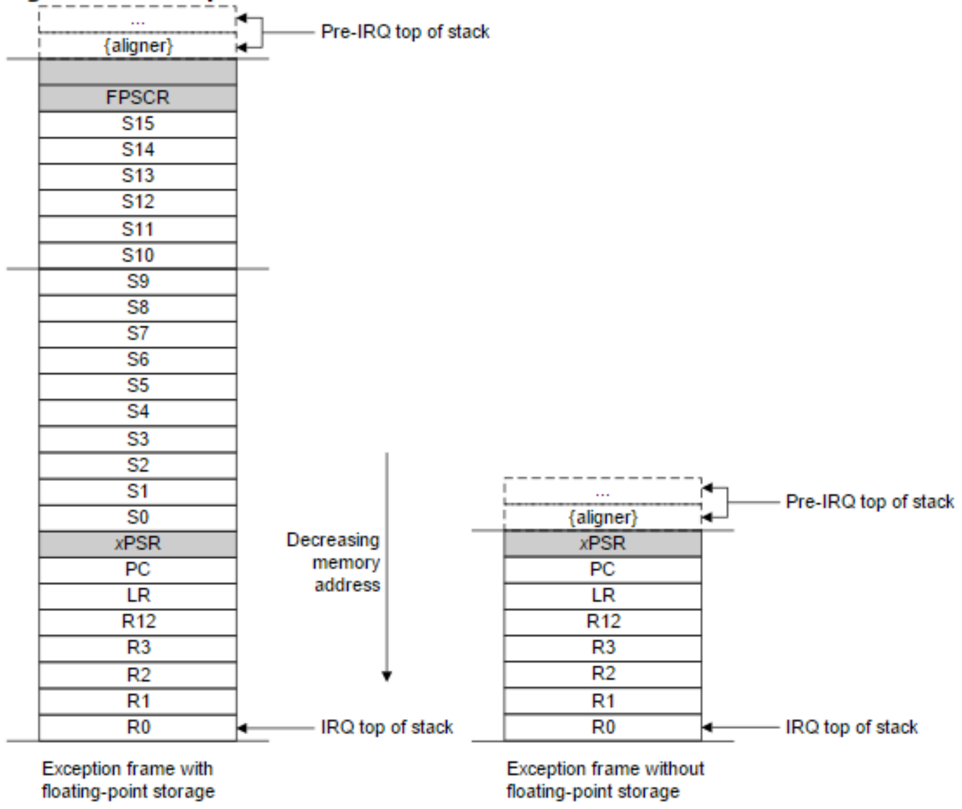
First set up your project as described in Lab 0 (including setting the scatter file).

The Cortex M4 processor supports two types of stack frames: one with floating-point storage and one without floating-point storage. The image below describes the differences between the two options. To make the code behave uniformly over all lab groups, make sure your implementation uses the exception frame WITHOUT floating-point storage. To do this, navigate to Target Settings » Target and make sure Floating Point Hardware is set to "Not Used", and **that the "Use MicroLIB" option is set**:



☐ Use Cross-Module Optimization
☑ Use MicroLIB ☐ Big Endian
Floating Point Hardware: Not Used

Figure 2.3. Exception stack frame


...
{aligner}
Pre-IRQ top of stack
FPSCR
S15
S14
S13
S12
S11
S10
S9
S8
S7
S6
S5
S4
S3
S2
S1
S0
xPSR
PC
LR
R12
R3
R2
R1
R0
IRQ top of stack
Decreasing memory address
Exception frame with floating-point storage
Exception frame without floating-point storage

Part 2: Adding the Files

This lab will build on your solution to Lab 3. You should use the following files from Lab 3:

3140.s – contains the assembly language definition of the timer interrupt and functions called when a process terminates.
3140_concur.c – contains the functions for allocating and freeing process stack space.

3140_concur.h – a header file with all function definitions listed.
utils.c – contains helper functions for setting up and using the LEDs.
utils.h – a header file with the function definitions for the functions defined in *utils.c*.
process.c – your implementation of basic round-robin scheduling.

The following files are additional for Lab 5:

realtime.h – a header file which contains the C definitions of variables and functions needed for real-time scheduling.
lab5_t0.c – a test program that runs a normal process with a real-time process.

Add them to your project by right clicking *Source Group 1* in the Project window and clicking on *Add Existing Files to Group 'Source Group 1'…* You may need to change the *Files of type:* in the file selection window to "All files (*.*)" to see all the files.

NOTE 2 If you are not confident in your implementation of Lab 3, then please contact the TAs at ece3140-staff@csl.cornell.edu to get a sample solution.

Part 3: Code Changes

Real-Time Support

In *process.c*, add the following include to the top of the file:

```
    #include "realtime.h"
```

You will need to modify the existing implementations of functions in *process.c* and additionally define the variables and implement the functions specified in *realtime.h*. You may need to add additional fields in `process_state` to allow processes to work with real-time scheduling.

Section III: Assignment

Part 1: Real-Time Clock

We will need a clock for our real-time scheduling algorithm. Use a different timer (other than PIT_0) to generate interrupts every millisecond (0.001s) and update our current time. For the purposes of this lab, all time will be relative to the point when `process_start()` is called, so your variables should correspond to the elapsed time since the call to `process_start()`. Implement this so that it does not impact the correct behavior of concurrent processes.

Create a global variable `current_time` of type `realtime_t` that corresponds to the elapsed time on the K64F in seconds and milliseconds. Logically, the current time in milliseconds is `1000*current_time.sec` + `current_time.msec`, and `current_time.msec` should always be less than 1000.

Part 2: Earliest Deadline First Scheduling

You will implement "earliest deadline first" (EDF) scheduling. The EDF scheduling algorithm selects the job with the earliest absolute deadline, provided that it is ready to start. This requires a modification to `process_select` and an additional function to create tasks with real-time constraints:

```
process_rt_create(void (*f)(void), int n, realtime_t *start,
realtime_t *deadline);
```

This creates a new process that should start after a `start` milliseconds have elapsed. It has a deadline of `deadline` milliseconds **relative** to `start`, **not relative** to `current_time`. `f` and `n` are the same as in `process_create()`.

Implement `process_rt_create` and modify `process_select` to implement two-level scheduling. Implement a real-time scheduling queue that has higher priority than the normal queue used for ordinary concurrent processes.

Your implementation should also keep track of the number of tasks that meet and miss their deadlines the global variables `process_deadline_met` and `process_deadline_miss` respectively.

Part 3: Testing your Implementation

We have provided a single test case for real-time scheduling. Your lab should pass this test and several tests that we will not provide.

You must also provide *two* test cases for your real-time implementation to demonstrate that your implementation works correctly. Each test case should focus on some non-trivial aspect of the real-time scheduling behavior. Name these files *test_r1.c and test_r2.c*.

Section IV: Extra Credit

Part 1: Periodic Tasks

Implement real-time scheduling of periodic tasks as well. Specifically, implement:

```
process_rt_periodic(void (*f)(void), int n, realtime_t *start,
realtime_t *deadline, realtime_t *period);
```

This creates a new process that should start after a `start` milliseconds have elapsed. It has a deadline of `deadline` milliseconds **relative** to `start`, **not relative** to `current_time`. The task is periodic, with period specified by `period`. `f` and `n` are the same as in `process_create()`.

The primary difference is that the task is periodic so once it terminates you must have it start at the appropriate time in the future. To do this correctly, you will need to make your own implementation of `process_stack_init` (e.g. `process_stack_reinit`) that reinitializes the stack and all the necessary context in the stack. Otherwise, the process will crash if you try to rerun it.

NOTE 5 We strongly recommend that your implementation of real-time scheduling (without periodic processes) works correctly even if your periodic process implementation does not; in other words, make sure that simply ignoring calls to `process_rt_periodic` would be a valid Lab 5 submission.

Implement `process_rt_periodic` and modify `process_select` to periodic real-time EDF scheduling.

Again, keep track of the number of jobs that have met and missed their deadlines.

Part 2: Testing your Implementation

We have not provided a test case for periodic scheduling. Your lab should pass several tests that we will not provide.

You must also provide *two* test cases that demonstrates that your periodic real-time implementation works correctly. Each test case should focus on some non-trivial aspect of the periodic real-time scheduling behavior. Name these files *test_p1.c and test_p2.c*.

Section V: Submission

The lab requires the following files to be submitted:

- **concurrency.zip**
  The entire implementation and test cases, including the files we have provided. Do not include µVision files.

- **report.pdf**:
  Provide a detailed description of your implementation of your real-time clock and your real-time scheduling, (max 5 pages in 11pt font and single line spacing. Also, don't do that little trick where you adjust the size of the periods or margins).

  We recommend using illustrations to describe your key data structures (processes, process queues, etc.) and key properties of the implementation that you use to ensure correct operation.

  You should also include a section on how you tested your implementation and describe the test cases you provide.

  We require that you describe in detail, in a separate section titled "Work Distribution", the following:

  a) How did you carry out the project? In your own words, identify the major components that constituted the project and how you conducted the a) design, b) coding, c) code review, d) testing, and e) documenting/writing. If you followed well-documented techniques (e.g., peer programming), this is the place to describe it.
  b) How did you collaborate? In your own words, explain how you divided the work, how you communicated with each other, and whether/how everyone on the team had an opportunity to play an active role in all the major tasks. If you used any sort of tools to facilitate collaboration, describe them here.

We encourage you to use meaningful variable and function names, comments, etc. to enhance code comprehensibility. All code files should be commented in a way that makes it clear why your program works.

All files should be uploaded to CMS before the deadline. Multiple submissions are allowed, but only the latest submission will be graded.