Lab 3: Concurrency

Goal:

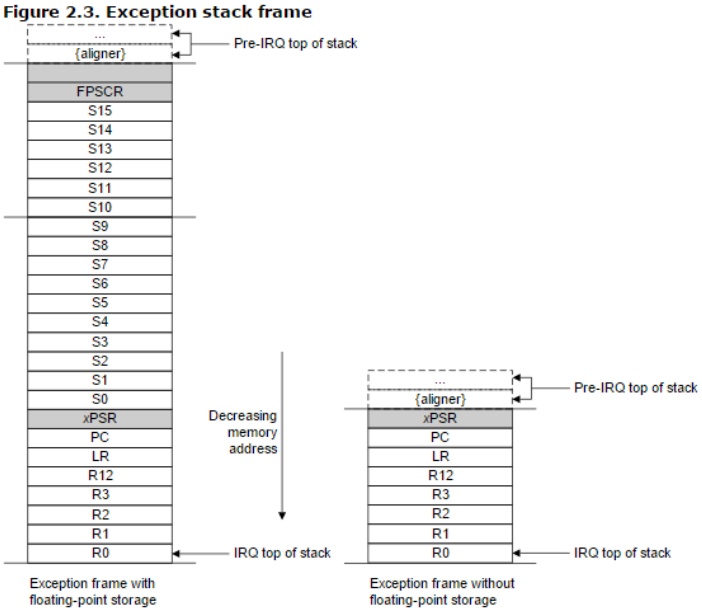
The purpose of this lab is to give you experience with I/O and interrupts on the FRDM-K64F microcontroller. To successfully complete this lab, you will need to understand the following:

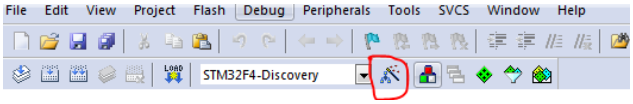
- Everything from Lab 2
- Concurrency and its implementation

We suggest that you first understand context switching 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.

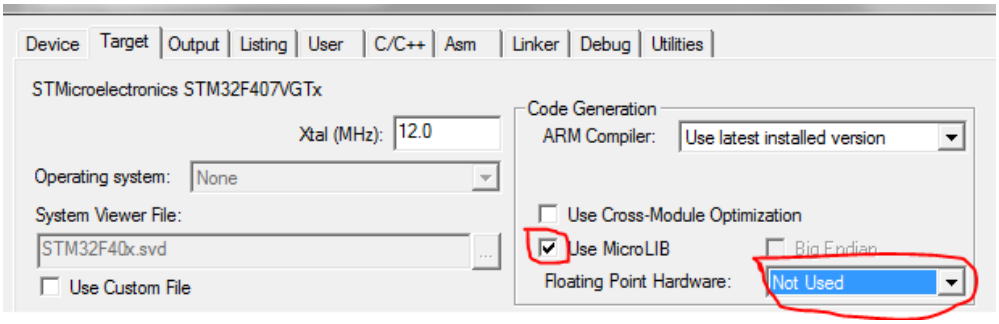
Part 1: Changes in Keil!!!

Emergency!!! The Cortex M4 processor supports two types of stack frames: one with floating-point storage and one without floating-point storage. The figure to the right describes the differences between the two options. In order to make Lab 3 behave uniformly over all lab groups, make sure your implementation uses the exception frame WITHOUT floating-point storage. In order to do this, navigate to Target Settings (shown in the picture below). Under the "target" panel, make sure Floating Point Hardware is set to "Not Used".





Target Settings:



Finally, check the "Use MicroLIB" box (shown above). Not checking this box will make your program stuck at "BKPT 0XAB" when debugging.

Part 2: Concurrency

In this lab, you will create a C program that uses concurrency. A sample program is provided as below:

```
void p1 (void)
{
   /* process 1 here */
   return;
}
void p2 (void)
{
   /* process 2 here */
    return;
}
   int main (void)
   {
      if (process_create (p1, 16) < 0){
         return -1;
      }
      if (process_create (p2, 16) < 0){
         return -1;
      }
      process_start();
      return 0;
   }
```

In the sample program, there're two processes p1 and p2 running concurrently. The main function creates, initializes and starts the two processes. A scheduler based on a timer switches the ownership of CPU between the two processes. When one of the processes owns the CPU, it performs tasks specified in `void p1 (void)` or `void p2 (void)`.

In this lab, you are required to implement some functions to complete this concurrency package. Detailed descriptions of these functions are as follows:

`int process_create (void (*f)(void), int n);`

This function creates and initializes a process that starts at function f, with an initial stack size of n. It should return -1 on an error, and 0 on success.

The implementation of this function requires that you allocate memory for a `process_t` structure, and you can use `malloc`() for this purpose. The state of the process can be initialized by calling the function

`int process_init (void (*f)(void), int n);`

We have provided this function for you, which allocates memory for a new stack of size at least n, initializes the stack for the process, and returns the value of the stack pointer. This function returns 0 if the memory allocation for the stack fails.

Finally, you should append the created process to the process queue (ready queue) to prepare for execution.

```
void process_start (void);
```

This function starts the execution of concurrent processes. As discussed in class, a context switch occurs on a timeout that is triggered by a timer interrupt. For this lab, we will use the periodic interval timer described in Lab 2. As such, the **`process_start`** function must:

1. Set up a period interrupt to be generated using the PIT[0] interrupt. The interrupt handler itself is provided for you in 3140.s, so you do *not* need to provide a C implementation of it.
2. Make sure all data structures you need are correctly initialized.
3. End with a call to `process_begin()`, which starts the first process.

The function `process_begin()` is provided for you in 3140.s in assembly.

```
unsigned int process_select(unsigned int cursp);
```

This function is called by the scheduler to select the next ready process from the ready queue. `cursp` is the stack pointer to the currently running process (0 if there is no currently running process). This function returns the stack pointer to the next ready process. If there is no process ready, it should return zero.
If the on-going process has not completed by the time **`process_select`** is called, the process should be appended to the ready queue so that it can resume later.

Besides the functions described above, your implementation must always maintain the global variable:

```
process_t *current_process
```

as the currently running process. This variable is set to NULL when a process terminates, and should also be NULL until `process_start()` is called.

There are three files provided to you:
3140.s – contains the assembly language definition of the timer interrupt and functions called when a process terminates.
3140_concurr.c – contains C definitions of functions for allocating stack space and initializing a process
3140_concur.h – a header file with all functions listed (including the ones that you must implement)

Part 3: Hints

We suggest that your implementation uses a data structure to maintain the process state of the following form:

```
typedef struct process_state {
      unsigned int sp;
      /* the stack pointer for the process */
      ...
} process_t;
```

In the presence of weak fairness of concurrent execution, a process may end up relinquishing the processor to allow another process to execute at any time. On the KL64F, we will achieve this by using timer interrupts.

However, it may be advisable to disable these interrupts temporarily to create larger atomic operations. The functions that support this are:

```
PIT->CHANNEL[0].TCTRL = 1;
PIT->CHANNEL[0].TCTRL = 3;
```

When a process terminates, the next process will be automatically selected for execution by calling your `process_select()` function. To help, we have provided some additional information about list data structures at the end of this document.

Part 4: What You Have to Do

The provided files should not be modified.
Instead, create a file `process.c` which contains your implementation of the `process_state` structure, and the functions `process_create`, `process_start`, and `process_select`, along with any other helper functions you may use (e.g. to manipulate the process queue).
You must also provide tests cases（at least two test cases） to demonstrate that your implementation works correctly. Name these files `lab3_tn.c`, where n is the test number. In the test cases, include as a comment which aspect of your code is being tested, and what is the expected behavior. The test case should include scenarios where you change the frequency of the timer interrupt to observe differences in the way the code is executed.

Documentation:
Provide a detailed description of your implementation of processes (max 5 pages in 11pt font and single line spacing – we will not read anything after page 5. 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, etc.) and key properties of the implementation that you use to ensure correct operation. This writeup should also demonstrate that you understand the files that were provided (3140.s, 3140_concur.c) in addition to your own implementation.

Test Cases:
We have posted a test case for your use. Your lab should pass all of these tests, and also a number of tests that we will not provide.

Submissions:
The lab requires the following files to be submitted.

1. Lab3.zip– which includes `process.c` and the test cases `lab3_tn.c` you created.
2. writeup.pdf – documentation of your implementation

The C files should be commented in a way that makes it clear why your program works.

Part 5: More hints. Lists

List data structures are commonly used to implement queues and stacks. A simple list data structure that holds a list of integers is given by:

```
struct mylist {
   int val;
   struct mylist *next;
};
```

You can use a single pointer to store the beginning of a linked list. Suppose this pointer is called `list_start`.

Initially, the list is empty. We need a special value of `list_start` that indicates this. In C, a common practice is to use the value of `NULL` (usually the integer 0) to indicate it. Hence, an empty list would be initialized by

```
list_start = NULL;
```

If `elem` is another `struct mylist` pointer, then we can insert it at the beginning of the list by the following operations:

```
elem->next = list_start;
list_start = elem;
```

The operation `elem->next` is shorthand for `(*elem).next`. We can traverse the list one item at a time as follows:

```
struct mylist *tmp;
for (tmp = list_start; tmp != NULL; tmp = tmp->next) {
     /* tmp->val is the integer of interest */
     /* do something here */
}
```

To insert an element at the end of the list is a bit more complicated. This is because there are two cases: (i) the list is currently empty; or (ii) the list has some items in it. If the list is empty, then the operation is easy. If the list is not empty, we need to traverse the list to find the last element in it, and then add the new one to the end of the list. The following code does this (assuming that `elem` is the new element to be added to the list):

```
struct mylist *tmp;
if (list_start == NULL) {
    list_start = elem;
    elem->next = NULL;
}
else {
    tmp = list_start;
    while (tmp->next) {
        /* while there are more elements in the list */
        tmp = tmp->next;
    }
    /* now tmp is the last element in the list */
    tmp->next = elem;
    elem->next = NULL;
}
```

We can remove the first element of the list in a straightforward way:

```
  if (list_start == NULL) {
    elem = NULL;
}
else {
    elem = list_start;
    list_start = list_start->next;
}
/* elem is the first element, and elem->val is the value. Normally
   we do not use elem->next at this point since we have removed it
   from the list. Sometimes programmers set elem->next = NULL just
   to be safe. */
```

A similar technique can be used to remove the last element of the list.
Remember, your implementation should not have to use assembly language other than the routines we have already provided.