AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
MECHATRONICS ENGINEERING DEPARTMENT
CREDIT HOURS PROGRAM

Spring Semester, 2024

CSE411: Embedded Systems

Lab (2): Creating Task Stacks for switching.

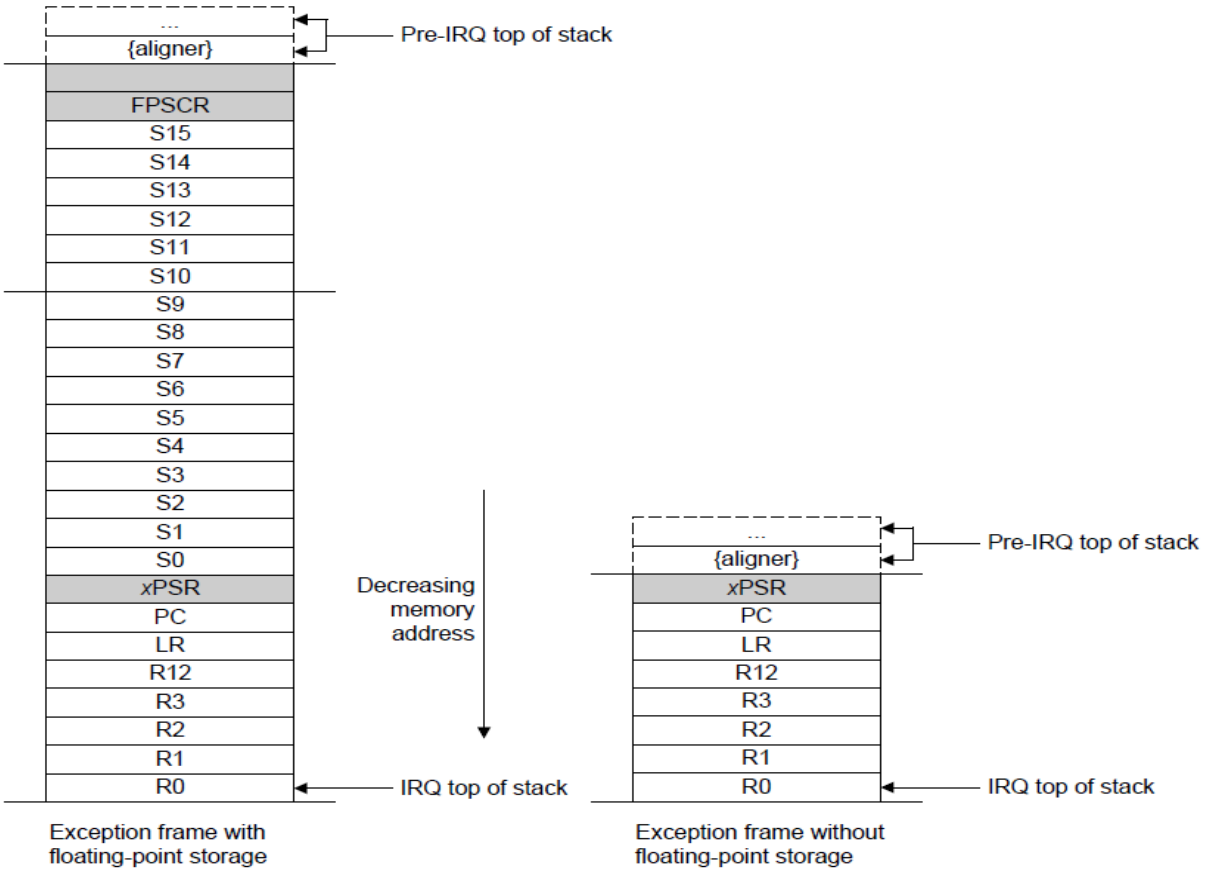
Goals of this Lab:

Real time operating systems are composed of various tasks. Tasks are infinite loop functions that serve certain functionalities. The operating system Kernel switches the processor resources (*processing time, hardware peripherals, etc*.) between tasks. The goal of this lab is to learn how to manually switch between two tasks.

Introduction:

What we did last lab which is manually switching by changing the value of the PC is not quite legal. It would also not work in larger applications. furthermore, it would cause some serious problems. We still need to have the code running automatically without interfering every time to switch among tasks.

To do so we need to look at what happens in the context switching exactly, and what happens exactly in the stacks. As per the TM4C datasheet please find the frames that are added to the stack while doing the context switch of an exception. The screenshot attached is valid if we turned off the Floating-point Unit (FPU) from the project settings.



ARM exception stack frame layout

NOTE: The stack pointer grows upward in the ARM which means the top of stack would be the address with the lower value.

Hint : You may need to flip the image above to match the stack if the values of the stack are put in ascending order.

Step 1: Use the code created in Lab 1

Open lab 1 progress. you have two tasks: one blinks the Red LED and the other blinks the Blue LED.

Step 2: Create a stack for each task:

Create two stacks to store the values for each task. Each stack can be created as an array of **40 uint32_t** elements. We also need a pointer that shall point to the top of the stack exactly like what the SP does.

```
/* Example for Task 1 */
uint32_t stack_RedBlinky[40];
uint32_t *sp_RedBlinky = &stack_RedBlinky[40]; /* The stack pointer is initialized to pint one word after the stack because the stack grows down
that is from the end of the stack array to its beginning */
```

Step 3: In the main function, fill the stacks created with the exception frame

According to the ARM exception frame layout attached above. We shall start from the high memory end of the stack because it grows from high to low memory.

1- Pre-decrement the pointer to get to the first address location, we then need to write the THUMB bit in the PSR register (Bit 24).
2- Pre-decrement the pointer again this time to write the PC. In C you can get the start address of a function using its name (ex: &Task1). You need to cast the pointer to fit inside the pointer (ex: (uint32_t) &Task1).
3- Initialize the other registers so we can observe that their values are copied to the stack.
4- Repeat steps 1 through 3 again, but this time for the second task. (Blinky LED 2)
5- Create an infinite loop at the end of the main function.

Step 4: Investigate the stacks in the memory view.

1- Look at how your stacks are allocated in memory and make sure you have the correct values in each place. Put both of your created stack pointers in the watch window.
2- Put a breakpoint in your systick Interrupt handler. Then wait till the debugger hits the breakpoint.
3- Manually change the value in the SP register to the stack pointer of Blinky1, remove the breakpoint, and watch as the debugger goes to the Blinky1 function.
4- Now, switch to Blinky2 and put back the breakpoint in the ISR. We first need to take the current value in the SP register and put it inside the created stack pointer for Blinky 1 because that value is now the top of the stack of Blinky1. After that we can put the value of Blinky2 stack pointer in the SP register.