Experiment 5: RTOS Design

Objectives:

- To gain experience writing your own RTOS on the TM4C123 Launchpad Development Board.
- To gain experience the Keil ARM development environment and debugging features.
- To learn more about concurrency, semaphores, and races conditions.

Reading:

- Tiva™ C Series Development and Evaluation Kits for Keil™ RealView® MDK (course website or http://www.ti.com/lit/ml/spmu355/spmu355.pdf
- Tiva C Series TM4C123G LaunchPad Evaluation Board User's Guide (course website)
- Tiva C Series TM4C123x ROM User's Guide (course website or http://www.ti.com/lit/pdf/spmu367)
- TivaWare Peripheral Driver Library User's Guide course website or http://www.ti.com/lit/spmu298)

Introduction:

In this experiment, you will experiment with the kernel of a simple RTOS and with some of the drivers available in TivaWare from TI by integrating TI drivers into an existing project.

Experiment:

1. First, download the project file from the course web site. This project has a working version of OS that uses the Edge triggered falling-edge triggered interrupts (which on your Launchpad board is Switch SW1). (NOTE: Because the push-buttons on the TM4C123 Tiva Development boards are not de-bounced, it is possible to trigger several interrupts per button press. Use the push button to confirm that your OS task switches between the various tasks. Instead of using the push-button for context switches, we would like to use the SysTick timer to initiate context switches.
2. First let us get the LEDs working so we can use these for outputs to confirm the OS is running without a serial program (e.g., putty). To gain experience with the TivaWare implementation let's use the ROM versions of the GPIO functions to drive the LEDs. In particular, we will use the following calls to enable Port F and to configure the three pins that have the LEDs attached on our development boards:

```
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
ROM_SysCtlDelay(1);
ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_3
|GPIO_PIN_2| GPIO_PIN_1);
```

Once the port is configured, we can use calls like the following to turn on and off the LEDs:

```
ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,
                 GPIO_PIN_1);     // Red LED on
ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 |GPIO_PIN_2 |
GPIO_PIN_1,
                 0);   //  All LEDs off

ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2 ,
~(ROM_GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2) );   //
toggle Blue LED
```

In order to use the ROM versions of the TivaWare routines you have to include header files that specify the function locations. In particular, the following must be included:

```
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
```

Also, add the following Defines under the C/C++ tab in Keil so that the TivaWare code knows which compiler “Real View Microcontroller Development Kit” and which target processor “TM4C123GH6PM” we are using:

```
rvmdk PART_TM4C123GH6PM TARGET_IS_TM4C123_RB1
```

Perhaps have each task toggle an LED to confirm that the OS is task switching.

3. To gain further experience with the TivaWare implementation we will again use the ROM versions of the SysTick functions to configure and enable the systick timer. For this part perhaps you can confirm the rate of interrupts with an ISR that toggles an LED with a scope.
E.g.,

```
void SysTick_Handler(void){
     ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2 ,
   ~(ROM_GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2) );
             // toggle PF2 (Blue LED)
}
```

A reasonable rate for context switches could be 1-50 ms.

Again, to use the ROM versions of the TivaWare routines you have to include header files that specify the function locations. In particular, the following must be included:

```
#include "driverlib/rom.h"
```

Then you have access to the functions:

```
void ROM_SysTickDisable (void)
void ROM_SysTickEnable (void)
void ROM_SysTickIntDisable (void)
void ROM_SysTickIntEnable (void)
uint32_t ROM_SysTickPeriodGet (void)
void ROM_SysTickPeriodSet (uint32_t ui32Period)
uint32_t ROM_SysTickValueGet (void)
```

and you could write a function like:

```
void SysTick_Init(){

  ROM_SysTickEnable();
  ROM_SysTickPeriodSet(16700000);
  ROM_SysTickIntEnable();

}
```

Once you have the Systick Interrupting at a reasonable rate, jelosASM.s file to include your SysTick ISR, e.g..:

```
SysTick_Handler
  push{r4-r11};save rest of state of the task switching out
  mov r0,sp
  bl Schedule  ; will call scheduler to select new task
  mov sp,r0          ; load new tasks sp
  pop {r4-r11}
  ldr lr,=INTERRUPT_LR
  bx lr    ; context switch!
```

4. Next we will add a very useful built-in command to the shell that will allow the shell user to see what tasks are currently running in the OS (and will allow you to get more familiar with the internal workings of the OS). The “ps” command will show you a listing of all the tasks currently running on the OS and status information about each.

```
os# ps
USER   TID  %CPU  STK_SZ   %STK  STATE      ADDR
root    1    9.3   4096    40.0   RDY      0x00000040
root    2    4.4   2000    88.0   RUN      0x000000A4
```

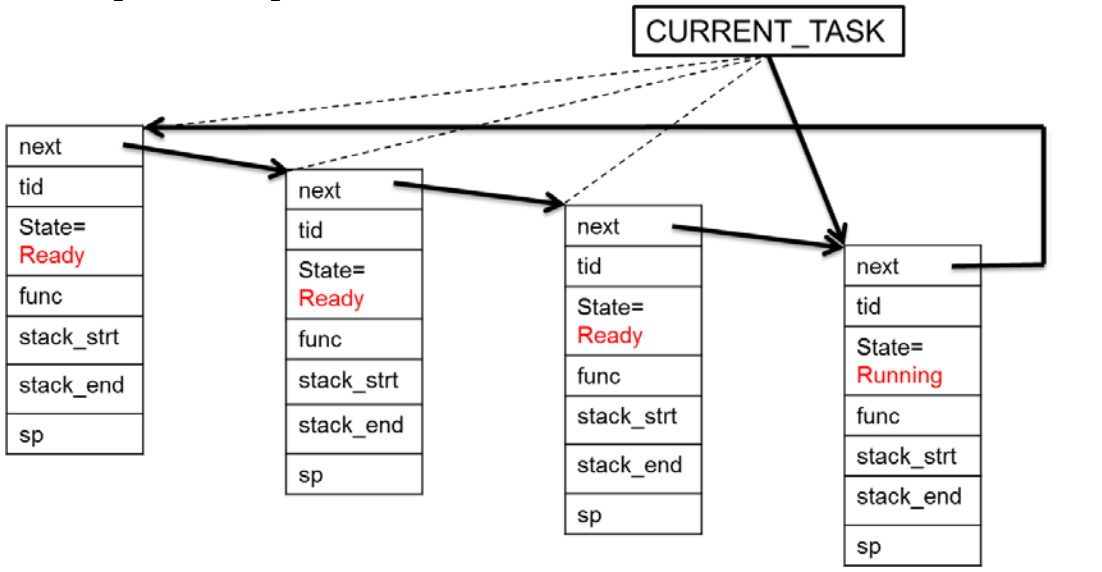
Where:
TID = task ID
%CPU = percent CPU used by this task
STK_SZ = stack size for the task
%STK = percent of available stack in use
STATE = task state

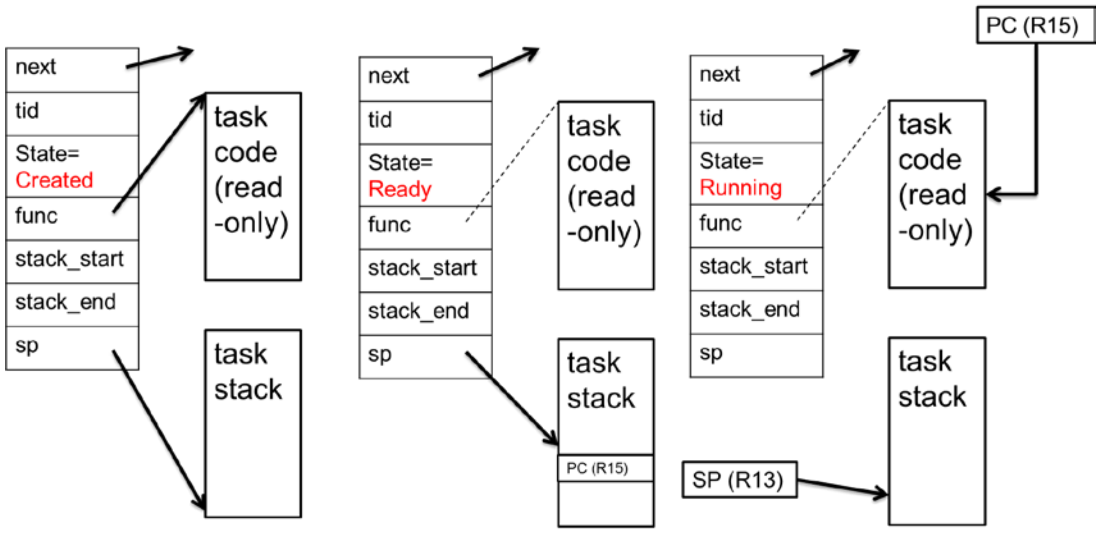
For reference, here is the declaration of the JELOS Tack Control Block

```
typedef struct TaskControlBlock
```

```
{
struct      TaskControlBlock *next; // pointer to next TCB
unsigned char tid;                  // task id
unsigned char state;                // task state
void  (*func)(void);    // ptr to function to call for that task
unsigned char     *stack_start;     // stack low value
unsigned char     *stack_end;       // stack high value
unsigned char     *sp;       //current value of the stack pointer
} TaskControlBlock;
```

And a figure showing how the TCBs are linked:





(a) TCB for a new task (b) TCB for a RDY task (c) TCB for a RUNning Task

Which of the ps fields are difficult/impossible to calculate? What would the OS need to calculate these? Demonstrate your ps function with various tasks running for the TA for a signature.

Next lets add a running processing time total to our tasks. Add a 32-bit unsigned int field to the TCB structure that will hold the total number of clock ticks that the task has run. Add code to the OS to set this counter to zero when a task starts and to update the field on context switches (perhaps using `ROM_SysTickValueGet()`). Then modify your "ps"command to report this information correctly.

5. Finally, we will add a semaphore capability to the OS that we can use to protect access to shared resources (e.g., the UART). For this first implementation we will use a "busy waiting" spin-lock semaphore. While a spin-lock is not the most efficient implementation, it is simple and will provide mutual exclusion. Basically we need two functions to signal and wait for a given semaphore (and perhaps a function "OS_Sem_Init()" to initialize the semaphore before we use it):

```
void OS_Sem_Signal(unsigned int *s)
{
   s = s + 1;
}

void OS_Sem_Wait(unsigned int *s)
{
   while (*s == 0){
         ;              // blocked
   }
   *s = *s – 1;
}

void OS_Sem_Init(unsigned int *s, unsigned int count)
{
   s = count;
}
```

However, the variable "s" itself is being shared and must be protected from race conditions. Since we are writing the code for our first semaphore our only hope is find some other way to ensure mutual exclusion within the semaphore code. We can do that by enabling and disabling interrupts. Add calls to EnableInterrupts() and DisableInterrupts() (both are provided in the startup) to the code above and implement a semaphore to protect the UART in the OS example code.

NAME:____________________

1-2: Instructor Signature:______________________ Date/Time:__________

3-4: Instructor Signature:______________________ Date/Time:__________

5: Instructor Signature:______________________ Date/Time:__________