# EEL 4745C: Microprocessor Applications 2
Lab ④: G8RTOS Dynamic Threads and LCD Interfacing
Fall 2022

**OBJECTIVES**
In this lab, we will:
- Write a library with extensive functions that allows interfacing a touchscreen color LCD.
- Incorporate dynamic and aperiodic event threads in our RTOS.
- Convert our round-robin scheduler into a priority scheduler.

**REQUIRED MATERIALS**

Hardware
- Tiva TM4C123GH6PM Launchpad
- LP3943 LED array modules (that are already) integrated to your daughterboard
- The ILI9341 LCD display provided to you.
- Laptop workstation with CCS setup

Software
- TivaWare and Board Support Package (BSP)
- Skeleton code provided for Lab 4

Documentations
- TM4C123GH6PM manual and datasheet
- LCD datasheets (same family)
  - ILI9341, ILI9325, TFT Manual
- LP3943 manual and datasheet
- Sensors Booster Pack datasheet
- Feel free to use books or browse the internet!

**IN-LAB REQUIREMENTS**
Lab 4 starts the week of (Oct 17-21); demos are due the following week (Oct 24-28). Late demos are effective 1 week afterwards with a 10% penalty. The solutions will be uploaded on Nov 4th.

**PRE-LAB QUESTIONS**
- What is SPI communication? When do you need to or should use it?
- What are the similarities/differences between: a periodic vs a dynamic thread?
- Think about the how to do these in your LCD:
  - Draw a line/rectangle/score-board.
  - Move an object with touch.

## **Part A:** Interfacing a touchscreen color LCD

For LCD interfacing, we have provided some driver functionalities: please thoroughly go through the `ILI9341_Lib.c` & `ILI9341_Lib.h` files in the Board Support package (BSP). Some functions are already implemented for your reference, and you will need to complete the rest.

Some of the important functions that are already implemented are as follows:

**`LCD_Init()`**
The provided LCD initialization function does most of the work for you, however you will need to initialize the SPI peripheral yourselves. Use *no pre-scaler* for maximum performance and enable the interrupt for the touchscreen if the user indicates to do so.

**`PutChar()`**
Outputs a character to the display at some coordinate. This utilizes the ASCII library provided to you.

**`LCD_Text()`**
Outputs a string to the display at some coordinate

**`LCD_WriteIndex()`**
Sets the address for the register we want to write to.

**`LCD_WriteData()`**
Writes 16-bit data to the register that is specified by the `LCD_WriteIndex()` function

**`LCD_Write_Data_Only()`**
Sends only data (useful for continuous transmission)

**`TP_ReadXY()`**
Reads the tappedX and Y coordinates from the LCD.

**Functions you will need to implement**

**`LCD_DrawRectangle()`**
Draw a rectangle with a specified color.

**`LCD_Clear()`**
Clear the screen with a specified color

**`LCD_SetPoint()`**
Draw one pixel with specified coordinate and color.

**`LCD_WriteReg()`**
Write data to the specified register.

**`LCD_SetCursor()`**
Place the cursor at the specified coordinate.

**`LCD_PushColor()`**
Set a pixel on the LCD to a specific color.

**`LCD_SetAddress()`**
Set the draw area of the LCD; please look into the manual/datasheet to correctly implement this.

*** Feel free to add any other functions if/as needed.*

### **Part B.1:** Convert Round-Robin Scheduler to Priority Scheduler

Instead of using a round-robin scheduler, you will introduce *priority* to our RTOS, where every thread has its own priority. You might need something like a background thread that always needs to execute as soon as possible. For example, later in this project, after tapping the LCD screen, a background thread might need to be executed first to update the global status as soon as possible. Also, you might want to lower the idle thread's execution frequency as much as possible too. You can accomplish all of this with different thread priorities.

***Hints:*** *You may follow these steps:*

- *Maintain a variable* `currentMaxPriority` *to store the priority level of the current running thread. Initialize* `currentMaxPriority` *to 256 (lowest priority); remember that the highest priority is 0.*
- *While scheduling another thread, check if its priority is less than* `currentMaxPriority`*.*
- *Lastly, modify G8RTOS_Launch to choose the thread with the highest priority to run first.*

### **Part B.2:** Dynamic Thread Creation and Destruction

Currently, we cannot add or kill threads once your OS has been launched. In order to accomplish this, we need to modify the TCB structure and also the `G8RTOS_AddThread` function: add a function to attain a thread's ID, and add two new functions to the scheduler:

- `G8RTOS_KillSelf()` and
- `G8RTOS_KillThread()`.

***Hints:***
*It is helpful to use the following:*

- *A boolean (say* `isAlive`*) to keep track of the status: alive or dead in the TCB.*
- *An integer (say* `threadID`*) and character array (say* `threadName`*): will be convenient to keep track of threads inside of the variable explorer and overall debugging process.*

So basically,

- The `isAlive` bit will indicate whether that thread is alive, or if it's been killed and no longer in the linked list of active threads.
- The `threadID` will allow every thread to have its unique ID so that the user can request the ID of the thread to be killed.

**Modifications to `G8RTOS_AddThread()`**
The `AddThread` function will now take in not only a thread's priority, but also its name to initialize. Since we want to be able to add a thread while our OS is running, we will need to enter a critical section and exit it prior to returning. Make sure that all `threadIDs` are unique and set the `isAlive` bit to true initially.

**`G8RTOS_KillThread(threadId)`**
This function will take in a `threadId`, indicating the thread to kill. ***Hint:*** *Think about how to take care of the boundary conditions (if no threads exist with that ID, only one thread running, etc.). Also think about which operations should be in the critical section.*

**`G8RTOS_KillSelf()`**
Very similar; this function will simply kill the currently running thread.

### **Part B.3:** Aperiodic Event Threads

Whenever you tap (use the LCD touchpad), we need to run an ISR is essentially an aperiodic event thread. To do this, we will need to initialize the appropriate NVIC registers accordingly. To add an aperiodic event, we provide it with a function pointer that will serve as the ISR, a priority, and the IRQ interrupt number:

- `G8RTOS_AddAPeriodicEvent(`
  `void (*AthreadToAdd)(void),`
  `uint8_t priority, int32_t IRQn)`

Make sure to verify that IRQn is less than the last exception (155) and greater than last acceptable user interrupt (0), and verify priority is not greater than 6, the greatest user priority number.

***Note:*** *To relocate an ISRs interrupt vector, the interrupt vector table must be relocated to SRAM. Depending on the compiler, this may or may not be done automatically. Therefore, to be compliant with all compilers, you want to relocate the interrupts vector to SRAM yourselves. You can do this in* `G8RTOS_Init()`*. The following code will relocate the vector table to address 0x20000000.*

```
uint32_t newVTORTable = 0x20000000;
uint32_t * newTable = (uint32_t
*)newVTORTable;
uint32_t * oldTable = (uint32_t *)0;

for (int i = 0; i < 155; i++)
{
    newTable[i] = oldTable[i];
}
HWREG(NVIC_VTABLE) = newVTORTable;
```

## General description of expected final product

- Program will launch with a blank screen waiting for a tap.
- Once touched, a ball (4x4 rectangle in our case) should be drawn on the screen with a random color; you may use the `time.h` library for randomness.
- Depending on the accelerometer X and Y values, the ball will change directions.
- To make it more interesting, every new ball created should have a random speed (just a scaling factor for its velocity).
- If any ball is touched, it should be deleted.
- There will be a maximum number of 20 balls allowed at any point of time.
- If a ball hits an edge, it should wrap around to the other side.

Initially, you will have the following threads active:

- Read Accelerometer: background thread
- LCD tap: aperiodic thread
- Wait for tap: background thread
    - Waits for a flag from ISR, reads touch coordinates, and then determines whether to delete or add a ball.
    - If a ball is to be created: write the coordinates to a FIFO and then create a Ball thread.
    - If a ball is to be deleted: wait for any semaphores the ball thread might be using and call `G8RTOS_KillThread` with the ball's `threadID`.
    - Delay for 500ms (try better values) to account for screen bouncing before checking the touch flag again.
- Idle thread: lowest priority thread
- Ball thread: background thread
    - Finds a dead ball and makes it alive; reads FIFO and initializes coordinates accordingly
    - Get `threadID` and store it; it is better if you use a struct to hold all information about a ball
        - Color, ID, position, velocity, etc.
    - Within while(1):
        - Move its position depending on velocity & acceleration
        - Then update the ball on screen and sleep for ~30ms (try better values).

>> See a sample demo video: **https://youtube.com/shorts/umWUxbx3qZc**