Lab 1 Graphics, LCD, ADC, Timer and Interpreter

Goals
- Introduction to Tiva TM4C123 LaunchPad,
- Interrupting serial port,
- Graphics LCD driver,
- Timer-triggered ADC driver,
- Periodic interrupts using the timer,
- Develop a command line interpreter.

Starter files **Located at http://users.ece.utexas.edu/~valvano/arm/**
The following Keil 5 projects are included in **`ValvanoWareTM4C123v6.zip`**
- **`ST7735_4C123`**
- **`PeriodicSysTickInts_4C123`**
- **`ADCSWTrigger_4C123`**
- **`UARTInts_4C123`** (includes FIFO code)
- **`PeriodicTimer0AInts_4C123`**
- **`RTOS_Lab1_Interpreter`** (starter project for Lab 1)

Data sheets (look for TM4C123 reference material on class website or Prof. Valvano's website)
- **`http://users.ece.utexas.edu/~gerstl/ece445m_s24/resources.html`**
- **`http://users.ece.utexas.edu/~valvano/Datasheets/`**

Background

The overall goal of the class will be to develop a real-time operating system. In this lab, however, you will familiarize yourself with the LaunchPad board, µVision development system and the TM4C123 ARM Cortex-M4 microcontroller. Most of the fundamental concepts in this lab should be review. Therefore, you will use this lab to explore the details of the development environment.

Look ahead to the next couple of labs. How you design this lab will simplify how you use these programs in subsequent labs. Do the lab in order. Do the preparation before coming to the first day of lab, do each step of the procedure before checking out. Read the entire lab assignment before starting the procedure, so you can gather the right data while you are doing the lab instead of at the end. Write the report the same day you finish checkout. Everything will be fresh in your mind and your lab will still be working so you can take meaningful data.

An important design step occurs in writing the header file for a driver. It is in the header file that you define the interfaces between software components. As part of the preparation in addition to the header files you will include rough pseudo code with descriptions of their approach to what you plan to write in the C files. As part of the preparation, you should have a plan of how you will complete the lab. The TA checks the preparation at the start of lab. This way the TA has an opportunity to set you on the right track by looking at what you have thought of so far.

Simply put, develop a TM4C123 project with 1) an interpreter running via the UART link to the PC, 2) an LCD that has two logically separate displays implemented on one physical display, 3) a periodic interrupt that maintains time, and 4) an ADC device driver that collects data using a second periodic interrupt. There are a lot of specifications outlined below, however, you are free to modify specifications as long as the above four components are implemented and understood. Part of the preparation is for you to clarify exactly what you will implement.

I recommend you avoid using SysTick, Timer0, Timer1, Timer 2 and Timer3 in this lab because subsequent labs will use these devices. Timer4 is used for the ADC task in the lab's starter code, and I otherwise recommend Timer5 or any of the wide timers (see **`WTimer0A.c`** included in the starter project) for this lab.

Prepreparation (do this individually)

0) Go to the ARM site to download the compiler to your laptop. Please get the newest compiler and install Keil uVision 5.x from here: http://www.keil.com/demo/eval/arm.htm (set company to "University of Texas at Austin" and devices to "TM4C123"). Download and install the Launchpad ICDI drivers from the TI website: http://www.ti.com/tool/stellaris_icdi_drivers, and install the ICDI debug adapter support add-on for Keil (see http://www.keil.com/support/docs/4196.htm). Finally, download and unzip the starter project files and board support package from Prof. Valvano's website: **`ValvanoWareTM4C123v6.zip`**.

1) Please review the style guideline presented in **`style.pdf`** and **`c_and_h_files.pdf`**.

2) Search through the **`UARTInts_4C123`** project to answer these questions about the UART port. The questions for prepreparation are for you to become familiar with the code and hardware needed in the remainder of the lab. They won't be turned in, so you don't have to write them down. However, if you feel more comfortable writing them down so that you have something to reference when we do the checkout for preparation, feel free to do so.

a) This example used UART0. What lines of C code define which port will be used for the UART channel?
b) What lines of C code define the baud rate, parity, data bits and stop bits for the UART?
c) Which port pins do we use for the UART? Which pin transmits and which pin receives?
d) Look in the **`uart.c`** driver to find what low-level C code inputs one byte from the UART port.
e) Similarly, find the low-level C code that outputs one byte to the UART port.
f) Find in the project the interrupt vector table. In particular, how does the system set the ISR vector?
g) This code **`UART0_ICR_R = UART_ICR_TXIC;`** acknowledges a serial transmit interrupt. Explain how the acknowledgement occurs in general for all devices and in specific for this device.
h) Look in the data sheet of the TM4C123 and determine the extent of hardware buffering of the UART channel. For example, simple microcontrollers like the MSP432 only have a transmit data register and a transmit shift register. So, the software can output two bytes before having to wait. The serial ports on the PC have 16 bytes of buffering. So, the software can output 16 bytes before having to wait. The MSP432 only has a receive data register and a receive shift register. This means the software must read the received data within 10 bit times after the receive flag is set in order to prevent overrun. Is the TM4C123 like the MSP432 (allowing just two bytes), or is it like the PC (having a larger hardware FIFO buffer)?

3) Search through the **`ST7735_4C123`** project to answer these questions about the LCD interface

a) What synchronization method do we use for the low-level command **`writedata`**?
b) Explain the parameters of the function **`ST7735_DrawChar`**. I.e., how do you use this function?
c) Which port pins do we use for the LCD? Find the connection diagram needed to interface the LCD.
d) Specify which other device shares pins with the LCD.

4) Search through the **`PeriodicSysTickInts_4C123`**, and **`ST7735_4C123`** projects to answer these questions about the SysTick interrupts.

a) What C code defines the period of the SysTick interrupt?
b) Look at these projects **`PeriodicSysTickInts_4C123`**, and **`ST7735_4C123`**. How does the software establish the bus frequency? Find the code that sets the **`SYSCTL_RCC`** and **`SYSCTL_RCC2`** registers. Look these two registers up in the data sheet. Look at these three projects to explain how the system clock is established. We will be running at 80 MHz for most labs in the class.
c) Look up in the data sheet what condition causes this SysTick interrupt and how we acknowledge this interrupt? In particular, what sets the COUNT flag in the **`NVIC_ST_CTRL_R`** and what clears it?

5) Look up the explicit sequence of events that occur as an interrupt is processed. Read section 2.5 in the TM4C123 data sheet (http://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf). Look at the assembly code generated for an interrupt service routine.

a) What is the first assembly instruction in the ISR? What is the last instruction?
b) How does the system save the prior context as it switches threads when an interrupt is triggered?
c) How does the system restore context as it switches back after executing the ISR?

Preparation (do this before your lab period)

1) Design an extended version of the device driver for the LCD so that there are two logically separate displays, one display using the top half and one display for the bottom half. A template for the extended LCD device driver to be used in this class is in the **`RTOS_Labs_common`** directory. This device driver will become part of your RTOS that will be further developed in the following labs. Add the solution to the file **`RTOS_Labs_common/ST7735.c`**. There should be 8 lines per display. The new command must have a prototype as follows:

```
void ST7735_Message (int device, int line, char *string, int32_t value);
```

where **device** specifies top or bottom, **line** specifies the line number (0 to 7), **string** is a pointer null terminated ASCII string, and **value** is a number to display. You may add other functions as you wish. In this lab, you may assume all public functions are called from the interpreter running as the main program; hence they need not handle pre-emption and reentrancy. However, in the next lab you will add semaphores so your LCD driver can be used by separate threads in a multi-thread environment. In labs 2 and beyond there will be multiple independent main programs, each performing output to its own LCD. For the preparation, design and write your implementation of **ST7735_Message** in the file **ST7735.c**. Your implementation will be debugged as part of the procedure.

2) Design a device driver for the ADC using software-triggered sampling. Sampling rates should vary from 100 to 10000 Hz, and data will be collected on any one of the ADC inputs ADC0 to ADC11. Feel free to use any existing code, as long as you completely understand how it works. A template for the device driver to be used with the labs in this class is in **RTOS_Labs_common**. The driver includes an **ADC.h** header file separating the public functions from the private functions. All public functions begin with an **ADC_**. Design and write the implementation file **ADC.c** for this driver. Your implementation will be debugged as part of the actual procedure. The driver provides the following two public functions for use in later labs:

```
// channelNum (0 to 11) specifies which pin is sampled with sequencer 3
// software start
// return with error 1, if channelNum>11,
// otherwise initialize ADC and return 0 (success)
int ADC_Init(uint32_t channelNum);

// software start sequencer 3 and return 12 bit ADC result
uint32_t ADC_In(void);
```

E.g., to take one sample from channel 0 using this driver, we can execute

```
ADC_Init(0);     // called once
Data = ADC_In(); // called whenever you need a new sample
```

E.g., to sample PE0, Channel 3 at 10 Hz, we define first a background task to run at 10 Hz

```
int32_t ADCdata,FilterOutput,Distance;
void DAStask(void){
  PF1 ^= 0x02;
  ADCdata = ADC_In();  // channel set when calling ADC_Init
  PF1 ^= 0x02;
  FilterOutput = Median(ADCdata); // 3-wide median filter
  Distance = IRDistance_Convert(FilterOutput,0);
  FilterWork++;        // calculation finished
  PF1 ^= 0x02;
}
```

We then initialized both the ADC and a periodic timer

```
int main(void){
  PLL_Init(Bus80MHz);
  UART_Init();       // serial I/O for interpreter
  ST7735_InitR(INITR_REDTAB); // LCD initialization
  LaunchPad_Init();  // debugging profile on PF1
  ADC_Init(3);       // channel 3 is PE0
  Timer4A_Init(&DAStask,80000000/10,1); // 10 Hz sampling, priority=1
  OS_ClearMsTime();   // start a periodic interrupt to maintain time
  EnableInterrupts();
  while(1){
    Interpreter();
  }
}
```

3) The skeleton for the OS to be developed in this class is provided on the **OS.h** and **OS.c** files in **RTOS_Labs_common**. As a first step in developing the OS, design and implement a basic timer driver and time servicing routines of the OS. Add a periodic timer interrupt to your **OS.c** file and use it to maintain the elapsed system time in ms. There are six 32-bit timers and six 64-bit timers you could use. Feel free to use any of the timers. However, please understand the periodic timer software, because as you progress through the class, depending on which features you use in your robot, you may have to move this time feature to another hardware timer. There is one public function that will reset the counter to 0 and start the periodic interrupt.

```
void OS_ClearMsTime(void);
```

A second public function will return the current global counter in ms.

```
uint32_t OS_MsTime(void);
```

Design and write the implementation of these OS time service routines. Your implementation will be debugged as part of the actual procedure.

Procedure (do this during your lab period)

1) Develop a main program that implements an interpreter that accepts commands over the serial port using interrupting I/O. You use the interpreter to test the various features of the lab. You can think of an interpreter as being similar to the Linux or Windows command line interface (shell), although your interpreter will be much simpler. For example, your interpreter needs to accept commands like "lcd_top" (write to the top LCD screen), "adc_in" (read a sample in from the hardcoded ADC channel and sequencer), etc. You could also just use commands like "1" or "2" or "a", etc., provided that a "help" command is provided to display what the commands mean. For checkout, you must present one demo. By interfacing with the interpreter, we should be able to test that your LED, ADC, UART, LCD and timer drivers all work. Implement commands that assist in debugging the LCD, ADC and OS time for this lab. For example, we should be able to type something like "lcd_top 0 hello" to print to the top LCD at line 0 the word hello. The interpreter accepts the command, parses it so that it can do the appropriate logic, then calls the LCD driver to send the information over UART to the LCD screen. The exact command syntax (names and arguments) are up to you, but the demo must demonstrate the functionality specified in this lab document. Provide for numeric input, numeric output and ease of use. There is no need to implement tab-completion or acceptance of backspace characters. For this lab, the interpreter will run in the main program as the only foreground task. It will later run as one of the main OS tasks. We will add commands as the semester progresses so make this interpreter flexible and extendable, because you will use it all semester as you add OS features. You can even use it to test the autonomous racing robot in Lab 6.

Your interpreter should use the **UART0int.c** driver provided in **RTOS_Labs_common**. In this driver, both input and output UART channels use interrupts and have software FIFOs so that the operating system can later block and unblock threads performing serial I/O. The serial channel is implemented as part of the USB link. You simply access the UART ports on the TM4C123 and run a terminal program like PuTTY or HyperTerminal to interact with your system through the command interpreter. There are two potential approaches, and you should choose the one you understand the best, because you will need to modify it in subsequent labs. The first option is to access the UART (via the driver and it's FIFOs) directly. Alternatively, you are allowed to use the **stdio** library and remap the serial stream to the UART. See the **fputc**/**fgetc** functions in the **OS.c** starter file.

2) Debug the LCD function **ST7735_Message**.

3) Debug the ADC functions **ADC_Init** and **ADC_In**. You may restrict your testing to one channel for the ADC, but in subsequent labs multiple channels will be used at a time. In particular, your robot may employ four distance sensors that utilize the ADC. Please limit the analog inputs to the ADC to 0 to 3.3V.

4) Debug the OS time routines **OS_ClearMsTime** and **OS_MsTime**. Look at disassembled code for your periodic ISR. Count the number of assembly instructions required to run one instance of this timer ISR. Assuming each assembly instruction is 2 bus cycles (25ns), approximate the overhead to maintain this time. I.e., let *n* be the number of assembly instructions and 1-ms be the interrupt period, then the *CPU utilization* for this OS task is

*n**25ns/1ms

5) Look at the dissembled code for one of the existing periodic interrupts, e.g., file **`Timer4A.c`,** included in the **`RTOS_Lab1_Interpreter`** starter project. In particular, look at the code for executing the C line **`(*PeriodicTask4)();`** in the Timer4A interrupt handler. This is called a *hook* and it is essential for later labs that you fully understand what this function call does and how it works (use of function pointers):

```
void Timer4A_Handler(void){
  TIMER4_ICR_R = TIMER_ICR_TATOCINT; // acknowledge TIMER4A timeout
  (*PeriodicTask4)();                // execute user task
}
```

6) *Triple toggle technique.* Run the **`RTOS_Lab1_Interpreter`** starter project and connect a scope or logic analyzer to PF1. Notice there are three toggles of PF1 in the **`DAStask`** function. Toggling three times allow you to measure both the time to execute the ISR, but also the time between executions. Using debugging instruments and a scope or logic analyzer, measure the actual time required to run the timer ISR. Measure the CPU utilization for this task, which is the time to execute ISR divided by time between executions.

Checkout (show this to the TA)

You should be able to demonstrate to the TA the technique you used to measure the overhead of running one of the periodic timer interrupt service routines. The successful completion of Lab 2 will depend on your knowledge of how interrupts are processed and how the serial port driver uses its two FIFO queues. Be prepared for questions addressing interrupts and the FIFO queue. Demonstrate each of the interpreter commands.

Deliverables (exact components of the lab report and lab submission)

A) Objectives (1/2 page maximum)
B) Hardware Design (none in this lab)
C) Software Design (software documentation in the report and check-in of all files into the submission repository)
   1) Low level LCD driver (**`ST7735.c`** and **`ST7735.h`** files)
   2) Low level ADC driver (**`ADC.c`** and **`ADC.h`** files)
   3) Low level timer driver (**`OS.c`** and **`OS.h`** files)
   4) High level main program (the interpreter)
D) Measurement Data
   1) Estimated time to run the OS periodic timer interrupt, CPU utilization
   2) Measured time to run the DAS periodic timer interrupt, CPU utilization
E) Analysis and Discussion (1 page maximum) This section will consist of explicit answers to these questions
   1) What are the range, resolution, and precision of the ADC?
   2) List the ways you can start the ADC conversion. Explain why you choose the way you did.
   3) You can measure the time to run the periodic interrupt directly by setting a bit high at the start of the ISR and clearing that bit at the end of the ISR. You could also calculate it indirectly by measuring the time lost when running a simple main program that toggles an output pin. How did you measure it? Compare and contrast your method to these two methods.
   4) Divide the time to execute once instance of the ISR by the total instructions in the ISR it to get the average time to execute an instruction. Compare this to the 12.5 ns system clock period (80 MHz).
   5) What are the range, resolution, and precision of the SysTick timer? I.e., answer this question relative to the **`NVIC_ST_CURRENT_R`** register in the Cortex M4 core peripherals.

Hints

1) It is appropriate to call existing software from the example files. You must, however, clearly document which code is copied, which code is modified, and which code is original.

2) Even though you are allowed to copy-paste software, there should be no magic in this class. In other words, you are responsible for understanding all the details of how your system runs.

3) Read ahead into Labs 2 and 3 to see how these drivers will be used. In particular, look at the user program that your Lab 2 RTOS will be running.

4) I suggest you use the starter project **`RTOS_Lab1_Interpreter`** for your Lab 1.

5) The **`RTOS_Labs_common`** folder contains a custom copy of the **`UART0int.c`** driver that we will use and modify throughout the labs in this class. This version of the driver uses dedicated software FIFO implementations from **`RTOS_Labs_common/FIFOsimple.c`**. You can find generic template-based FIFO implementations using C pre-processor macros in the **`RTOS_Labs_common/FIFO.h`** file.

6) The TM4C123 has 8 UARTs. UART0 (PA1, PA0) is connected to the PC through the USB cable.

7) If your board has not arrived yet, see your TA immediately, because none of the labs in this class can be performed in simulation. Simulation can only be used if you restrict your code to UART0 busy-wait, ADC0 software triggered busy-wait, and 32 bit periodic interrupts on Timer0, Timer1, Timer2 and Timer3.

8) If you have a LaunchPad, but no ST7735R LCD, you can perform Labs 1, 2, 3, 4 and 5 using any LCD (in particular we have some Nokia5110 LCDs to lend). You will find Nokia software drivers in the **inc** folder.

9) If you don't use **stdio** redirection to the UART for your interpreter, you can also redirect the standard output to the LCD display (or later a file after Lab 4), e.g. to support **`printf()`** style debugging. You can possibly also support different redirection targets via a global configuration switch/variable. The **`fputc`**/**`fgetc`** functions in the provided **`OS.c`** starter code show an example of toggling between UART redirection and redirecting to a file (needed for Lab 4).