Lab 1b Serial, oLED, ADC, Timer and Interpreter

Goals
- Introduction to Stellaris LM3S8962 board,
- Interrupting serial port,
- oLED driver,
- ADC driver,
- Periodic interrupts using the timer,
- Develop a command line interpreter.

Starter files
- **`PeriodicSysTickInts_8962.zip`** project (*Valvanoware*)
- **`OLED_8962.zip`** (*Valvanoware*)
- **`ADCT0ATrigger_xxx.zip`** (*Valvanoware*)
- **`uart2_1968.zip`** (*Valvanoware*), if you use this you need to convert to 8962
- **`lm3s8962.h`** (*Stellarisware\inc*)
- **`FIFO_xxx.zip`** (*Valvanoware*)
- **`UART_echo`** project (*Stellarisware\boards\ek-lm3s8962\uart_echo*)
- **`Timers`** project (*Stellarisware\boards\ek-lm3s8962\timers*)
- **`adc.c adc.h`** (*Stellarisware\driverlib*)
- **`systick.c systick.h`** (*Stellarisware\driverlib*)
- **`timer.c timer.h`** (*Stellarisware\driverlib*)
- **`uart.c uart.h`** (*Stellarisware\driverlib*)
- **`uartstdio.c uartstdio.h`** (*Stellarisware\utils*)

TI Data sheets
- DS-LM3S8962 Stellaris® LM3S8962 Microcontroller DATA SHEET
- UM-COREISM Cortex-M3 Instruction Set User’s Manual
- PB-LM3S8962 Stellaris® LM3S8962 Evaluation Board Product Brief
- EK-LM3S8962 Stellaris® LM3S8962 Evaluation Board User’s Manual

TI application notes
- AN01280 UART (spma032)
- AN01247 ADC (spma028)

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 LM3S8962 board, μVision development system and the LM3S8962 Arm Cortex-M3 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're 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.

Prepreparation (do this individually)

0: Please review the style guideline presented in **`style.pdf`** and **`c_and_h_files.pdf`**. Download and install the Keil's uVision4 compiler and TI's StellarisWare® software package as described in the course descriptor. Download and review the data sheets and application notes.

1: Search through the **`UART_echo`** project to answer these questions about the UART port
a) What line of C code defines 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 are used 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 know **`UARTIntHandler`**?
g) The function **`UARTIntClear`** acknowledges a serial transmit interrupt. Find both the C code and the assembly code for this function and explain how the acknowledgement occurs. Give the assembly code for this operation and explain how the acknowledgement occurs. In particular, explain what is in Registers **`r1`**, **`r0`** and **`lr`**. You can see the assembly code by debugging the system in the simulator.
h) Look in the data sheet of the LM3S8962 and determine the extent of hardware buffering of the UART channel. For example, the 9S12 transmitter has 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 9S12 receiver 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 RDRF flag is set in order to prevent overrun. Is the LM3S8962 like the 9S12 (allowing just two bytes), or is it like the PC (having a larger hardware fifo buffer)?

2: Search through the **`OLED_8962.zip`** project to answer these questions about the oLED interface
a) What synchronization method is used for the low-level command **`RITWriteData`**?
b) Explain the parameters of the function **`RIT128x96x4StringDraw`**. I.e., how do you use this function?
c) Looking at the schematics of the LM3S8962 evaluation board, which port pins are used for the oLED? Give a list of I/O pin assignments for the oLED.
d) Specify which other device on the board shares pins with the oLED.

3: Search through the **`LCD_Blinky`** project to answer these questions about the SysTick interrupts.
a) What C code defines the period of the SysTick interrupt?
b) The LM3S8962 runs at 8 MHz on the **`UART_echo`** project and 50 MHz on the the **`LCD_Blinky`** project. Find the **`RCC`** and **`RCC2`** registers in the data sheet. Look at the **`SysCtlClockSet`** function in the **`sysctl.c`** library file to explain how the system clock is established. We will be running at 50 MHz for most labs in the class.
c) Look up in the data sheet what condition causes this SysTick interrupt and how this interrupt is acknowledged?

4: Look up the explicit sequence of events that occur as an interrupt is processed. Read section 2.5 in the LM3S8962 data sheet. 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: Extend the device driver for the oLED so that there are two logically separate displays, one display using the top half and one display for the bottom half. There should be at least 4 lines per display. The new command will have a prototype of something like

```
void oLED_Message (int device, int line, char *string, long value);
```

where **`device`** specifies top or bottom, **`line`** specifies the line number (0 to 3), **`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. Graphics will be used in Lab 4, when we make a digital scope and spectrum analyzer, so you should import and modify the files **`rit128x96x4.h`** and **`rit128x96x4.c`** from the book web site (*http://users.ece.utexas.edu/~valvano/arm/*). However, in the next lab you will add semaphores so your oLED 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 oLED. For the preparation, add to the `rit128x96x4.h` header file prototypes for your public functions. All your public functions must begin with an `oLED_`. Your implementations will be written and debugged in the file `RIT128x96x4.c` as part of the procedure.

2: Design a device driver for the ADC. Sampling rates should vary from 100 to 10000 Hz, and data will be collected on any one of the ADC inputs ADC0 to ADC3. You are expected to use the existing driver functions. You are free to use whatever synchronization mode you wish. For example, you may software start the ADC selecting which channels to sample, wait for conversion, then return the digital results. In lab 2 you perform real-time sampling by triggering the conversion from timer0 and interrupting on ADC completion. In this way, there will be no sampling jitter. For the preparation, write an `ADC.h` header file separating the public functions from the private functions. All public functions must begin with an `ADC_`. The implementation file `ADC.c` for this driver will be written and debugged as part of the procedure. One possibility for the driver is (the first two are sufficient for Lab 1)

```
int ADC_Open(unsigned int channelNum);
unsigned short ADC_In(void);
int ADC_Collect(unsigned int channelNum, unsigned int fs,
       unsigned short buffer[], unsigned int numberOfSamples);
```

E.g., to take one sample from channel 0, we execute

```
  ADC_Open(0);
  Data = ADC_In();
```

E.g., to collect 64 samples from channel 1 at 10 kHz, and store the results in `DataBuffer`, we execute

```
  ADC_Collect(1, 10000, DataBuffer, 64);
  int ADC_Status(void);
```

The function `ADC_Collect` starts the conversion and the function `ADC_Status` returns 0 when it is done. Read the errata about timer-triggered ADC sampling. YOU MUST USE 16-BIT MODE FOR TIMER-TRIGGERED ADC SAMPLING. YOU CANNOT USE TIMER3 to trigger the ADC.

Procedure (do this during your lab period)

1: Develop a main program that implements an interpreter using the serial port and interrupting I/O. Both input and output channels must interrupt. Using the hardware FIFOs is optional, but you will need two software FIFOs so that the operating system can 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 LM3S8962 and run a terminal program like HyperTerminal to interface with your system. Commands will be added as the semester progresses so make this interpreter flexible. Provide for numeric input, numeric output and ease of use. The specific commands needed for this lab are those that assist in debugging the oLED, ADC and timer2 drivers. There are three potential approaches, and you should choose the one you understand the best, because you will need to modify it in subsequent labs. First, you can start with either the `UART_echo` or `UART2` projects, and add a receive FIFO and a transmit FIFO. To perform decimal output you could write your own or use `sprintf` like the `LCD_Blinky` project. The second approach is to add fifos to the utility `uartstdio`. A third approach is to use the `stdio` library and remap the serial stream to the UART. See the `retarget.c` file in the Keil\ARM\Boards\Keil\MCBSTM32\Blinky project included in the Keil installation. In particular, you need to create a `fputc` function like this

```
int fputc(int ch, FILE *f){
  Serial_OutChar(ch);
  return (1);
}
int fgetc (FILE *f){
  return (Serial_InChar());
}
int ferror(FILE *f){
  /* Your implementation of ferror */
  return EOF;
}
```

2: Implement and test the oLED driver.

3: Implement and test the ADC driver. You may restrict your testing to ADC0, but in subsequent labs ADC1 ADC2 and ADC3 will also be used. In particular, your robot will employ multiple distance sensors that utilize the ADC. Please limit the analog inputs to the ADC to 0 to 3V.

4: Design and test a system time driver using timer2 interrupts. Feel free to use any of the four timers except the one you plan to use for hardware-triggered ADC sampling (**`ADC_Collect`**). The overall operation will be to execute a software task at a periodic rate. You will pass a function pointer into this driver during initialization. E.g.,

```
int OS_AddPeriodicThread(void(*task)(void),
    unsigned long period, unsigned long priority);
```

where **`task`** is a pointer to the function to execute every **`period`** milliseconds, and **`priority`** is the value to be specified in the NVIC for this thread. This period could vary from 1 to 100 msec. A 32-bit global counter will also be incremented at this rate. In order to simplify transition into Lab2, I suggest you name this implementation file **`OS.c`**, with a header file **`OS.h`**. One public function will reset the 32-bit counter to 0.

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

A second public function will return the current 32-bit global counter.

```
unsigned long OS_MsTime(void);
```

Looking at disassembled code for the ISR count the number of instructions required to run once instance of the timer2 ISR. Specify the user task is a do-nothing function like this

```
void dummy(void){};
```

You will not be able to estimate the execute time from the assembly code because so many operations on the Arm architecture occur in parallel. Using debugging instruments and a scope or logic analyzer measure the actual time required to run the timer2 ISR. In the next lab, you will also need a record time function with a resolution on the scale of μsec. See the Lab2 starter file **`OS.h`**, and look particularly at the functions **`OS_Time`**.

Deliverables (exact components of the lab report)

A) Objectives (1/2 page maximum)
B) Hardware Design (none in this lab)
C) Software Design (printout of these software components)
  1) Low level oLED driver (**`rit128x96x4.c`** and **`rit128x96x4.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 periodic timer2 interrupt
  2) Measured time to run the periodic timer2 interrupt
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 the ADC conversion can be started. Explain why you choose the way you did.
3) The measured time to run the periodic interrupt can be measured directly by setting a bit high at the start of the ISR and clearing that bit at the end of the ISR. It could also be measured 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.
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 20 ns system clock period (50 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 M3 core peripherals.

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 the timer2 ISR. 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.

Hints

1) It is appropriate to copy-paste 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 modify the starter project **UART2_1968** into your Lab 1 solution. You should first convert it to the 8962. Add to this project the code from **OLED_8962**. You can find this FIFO in FIFO_xxx.zip

```
#define AddFifo(NAME,SIZE,TYPE, SUCCESS,FAIL) \
unsigned long volatile PutI ## NAME;  \
unsigned long volatile GetI ## NAME;  \
TYPE static Fifo ## NAME [SIZE];      \
void NAME ## Fifo_Init(void){         \
  PutI ## NAME= GetI ## NAME = 0;     \
}                                     \
int NAME ## Fifo_Put (TYPE data){     \
  if(( PutI ## NAME - GetI ## NAME ) & ~(SIZE-1)){  \
    return(FAIL);      \
  }                    \
  Fifo ## NAME[ PutI ## NAME &(SIZE-1)] = data; \
  PutI ## NAME ## ++;  \
  return(SUCCESS);     \
}                      \
int NAME ## Fifo_Get (TYPE *datapt){  \
  if( PutI ## NAME == GetI ## NAME ){ \
    return(FAIL);      \
  }                    \
  *datapt = Fifo ## NAME[ GetI ## NAME &(SIZE-1)];  \
  GetI ## NAME ## ++;  \
  return(SUCCESS);     \
}

AddFifo(Tx,32,unsigned char, 1,0)
AddFifo(Rx,16,unsigned char, 1,0)
AddFifo(Data,8,unsigned short, 1,0)
int test(void){ int r; unsigned char data; unsigned short sdata;
  RxFifo_Init();
  data = 3;
  r = RxFifo_Put(data);
  r = RxFifo_Get(&data);
  TxFifo_Init();
  r = TxFifo_Put(data);
  r = TxFifo_Get(&data);
  DataFifo_Init();
  sdata = 1000;
  r = DataFifo_Put(sdata);
  r = DataFifo_Get(&sdata);
  return r;
}
```

1) I suggest you use one of the valvanoware starter files to morph into the Lab1,2,3 sequence
http://users.ece.utexas.edu/~valvano/arm/ConvertLM3S1968intoLM3S8962.pdf
http://www.youtube.com/watch?v=l6IzNHVjohU how to change the name of a project
good starter files to choose from

http://users.ece.utexas.edu/~valvano/arm/OLED_8962.zip
http://users.ece.utexas.edu/~valvano/arm/PeriodicSysTickInts_8962.zip
however you are free to build your labs on top of stellarisware starter examples

2) The LM3S8962 has 3 uarts, other microcontrollers have up to 8. So, my examples tend to be simple and explicit for one microcontroller, and stellarisware are complex and general for the hundreds of different versions of the LM3S and LM4F microcontrollers available. It is your choice to decide how you wish to do the labs, as long as you understand how the I/O works. In this class the uart will always be UART0 and it will always be connected to the PC through the USB cable