EEL 4744 - Fall 2014

Revision **4**

Lab 5: Interrupts and Serial Communication

OBJECTIVES

- Explore and understand microprocessor interrupts. In part A of this lab, you will use XMEGA external interrupt system.
- Learn how to utilize asynchronous serial communication. In part C of this lab, you will use XMEGA's USART system to transmit/receive characters to/from computer terminal as well as write interrupt driven USART routines and explore multitasking.

REQUIRED MATERIALS

- EEL 4744 Board kit and tools
- DAD (Diligent Analog Discovery) kit
- No new hardware
- XMEGA documents
  - doc8331, section 12, 13, and 23
  - doc8385, section 33 (pinout and pin functions)

YOU WILL NOT BE ALLOWED INTO YOUR LAB SECTION WITHOUT THE REQUIRED PRE-LAB.

PRELAB REQUIREMENTS

REMEMBER:
You must adhere to the *Lab Rules and* Policies document for **every** lab.

PART A: EXTERNAL INTERRUPTS

In Lab 2 of this course, you added switches to your board. You had to continuously poll the input port to know the state of the switches. Polling wastes processor clock cycles and results in a slow response time if the processor is engaged in another instruction execution task. What if you wanted the processor to respond instantly to any changes on an input switch? The answer is to utilize external interrupts.

1. Read doc8331, sections 13.5-13.7 regarding external interrupts on the XMEGA chip. Then read section 13.13, which discusses the various registers you will need to use.
2. Read doc 8331, chapter 12 about the Programmable Multilevel Interrupt Controller (PMIC). Pay close attention to section 12.5 regarding interrupt levels.
3. Connect a single switch to any unused pin of your choice (PORTs C-F).
4. You will initialize XMEGA to trigger an interrupt when a **falling edge** is detected on the input. The interrupt service routine (ISR) should only execute when the pin value changes from to VCC to GND. But because we are unable to debounce the switches on our board, the ISR will fire on both edges. Therefore, inside the ISR you will have to test the value of the pin and only complete the desired function if the pin is the correct level.
5. Your initializations should proceed as follows:
   a. Select an interrupt priority level in the interrupt control register (PORTx_INTCTRL) for a port of your choosing.
   b. Select a pin on that same port as a source for the interrupt in one of the interrupt mask registers (PORTx_INTnMASK).
   c. Be sure to select the data direction for the input pin. Do not make assumptions.
   d. Select the input/sense configuration for the pin you selected in the PORTx_PINnCTRL register. Make sure you select sense falling edge.
   e. Turn on appropriate PMIC interrupt level in the (PMIC_CTRL) control register.
   f. In any interrupt driven application, the global interrupt flag should be the last thing you set during initializations. Simply use the `sei` instruction to set the global interrupt flag to enable the interrupt.
6. The processor needs to know what code to execute when an interrupt occurs. After an interrupt occurs, there is a specific address to which the processor will jump. This address is called the interrupt vector address. The name of the vector for this interrupt is PORTx_INTn_vect where x and n are defined by you, based on the interrupt pin that you selected. This vector's value is defined in the avr/io.h file and in the "Our ATxmega128A1U Register Descriptions" file from the Software/Docs section of the class website.
7. Write an ISR to display a count of how many times interrupt has been executed, on your LEDs. (Be sure to initialize the count to zero and LED pin directions to output~~, in your .c file~~.) The label of the ISR **must** be the same as the name of the vector that represents it (PORTF_INT0_vect, for example), and end with a `reti` instruction. The `reti` instruction is a special return statement reserved only for interrupt service routines and is (slightly) different from a `ret` instruction. By naming your ISR the same as the vector, the compiler knows how to put the appropriate `rjmp/jmp` instruction in the vector table, based on where it locates the actual routine in memory. With this jump instruction, the processor knows what to execute when your external interrupt occurs.
8. You must, as always, put the origin of your program at 0x200 or beyond. This is necessary because the interrupt vectors, which we will use from this lab onward, are located in the lower 506 addresses of program memory (0x000-0x1FA). You would also still need an appropriate `jmp` or `rjmp` instruction at `.org 0x0`. The 0x0 address is the reset interrupt vector, which cause your program to start executing at the address corresponding to the label for this `jmp` or `rjmp` instruction.

Lab 5: Interrupts and Serial Communication

9. Make sure your interrupt routine is working correctly by putting a breakpoint inside of it. The interrupt should only fire on a **falling** edge of the input pin.
10. Real switches bounce, so the count will probably be wrong! **Record the number of bounces** that you see when the switch goes from low to high and from high to low. (Do this a few times to see if it is always the same.). Put the shortest possible delay routine inside your ISR to prevent the bouncing on low to high transition from causing a problem.

PART B: ASYNCHRONUOUS SERIAL COMMUNICATION

Asynchronous serial communication can be done entirely in software by "bit banging." For example, if you understand the format of asynchronous serial data (see class notes), and you want to send data, then all you have to do to transmit a block of data on a GPIO pin. You would output the following bits at the required frequency or baud rate: start bit, 8 data bits (least significant bit first), and then a stop bit. This bit banging is **NOT** part of the lab requirements but is always an option if the processor has no support for serial communication. Similarly, a receiver can be made by sampling inputs at the appropriate times.

PART C: XMEGA USART System

The XMEGA has several universal synchronous asynchronous receiver transmitter (USART) peripherals. One of them, PORTC's USARTC0, is connected to the USB port labeled J3 on your board through a FTDI USB bridge chip (FT232RL). See the uTinkerer schematic for more information. Determine which pins on PORTC are used for USARTC0 using table 33-3 in doc8385 (alternate pin functions). (You won't need this info, but include this answer in your prelab.)

In this part of the lab you will write a program to send and receive data between uTinkerer board and your laptop using USARTC0.

You will need to install a terminal program on your laptop to communicate with your board. Some examples are Bray Terminal (also known as Br@y++ Terminal), PuTTY, X-CTU, RealTerm, and HyperTerminal. Install and familiarize yourself with one of them. (A description on how to use Bray Terminal is given in the Appendix.) The example program used in class is available on our web site (SCI_Polling.asm) and can be used as a starting point for your program.

To successfully complete this portion of the lab, I suggest that you create the subroutines/functions described below.

1. *USART Initialization function* (USART_INIT). This function will take care of all of USART initializations.
   a. Set the data direction of the PORTC USART bits to the corresponding values for Tx and Rx.
   b. Set the baud rate to 28,800 Hz (bits per second) by storing the appropriate value in the baud rate registers. [You can use a program at http://tinyurl.com/lhunuy2 to verify your baud rate calculation, but be sure that you know how to calculate these for your lab quiz and exams.]
   c. Set the USART for asynchronous mode, 8 data bits, 1 start bit, and 1 stop bit. Turn off wired-or, loop mode and parity.
2. *Character output subroutine* (OUT_CHAR). This subroutine will output a single character to the transmit pin of the XMEGA's USART system.
   a. Check if the previously transmitted character has been completed; if not, keep checking until it has been completed.
   b. Transmit the character passed to the subroutine.
3. *String output subroutine* (OUT_STRING). This routine will output character strings stored in program ~~data~~ memory.
   a. Read the character pointed to by Z ~~X (or Z)~~ and increment the pointer.
   b. If this character is not null, call OUT_CHAR, and repeat for each non-null character that follows; otherwise return from the subroutine.
4. *Character input subroutine* (IN_CHAR). This subroutine will receive a single character from the receiver pin of the XMEGA's USART system.
   a. Check if a character has been received, if not, keep checking until it has been received.
   b. Read the character from the receive buffer

Write an interactive program to display your favorite activities on Bray Terminal (or any other terminal of your choice). The menu you will create should look like this:

your_name's favorite:
0: Book
1: Movie
2: Sport
3: TV show
4: Food
5: Display menu
ESC: exit

When your program first starts it should display the above menu using OUT_STRING subroutine and then wait for an incoming character. When a 0, 1, 2, 3, 4 or 5 is received, output the corresponding message, such as "Cookie Monster's favorite food is cookies" and redisplay the menu. If the received character is an escape character, output "Done!" and terminate execution. If any other character is received, ignore it. For readability, add a blank line between every message.

You will need two special characters to cause the curser to move down a line. These two characters are carriage return (CR) and line feed (LF). The carriage return will move the cursor all the way to the left and line feed will move the cursor to the next line. These characters should

Lab 5: Interrupts and Serial Communication

occur in order at the end of every line. In addition, an escape character (ESC) will be used to terminate your program (Note: On at least one tablet PC last year, the ESC key code did NOT return a $1B. If this is the case for your computer, just pick another key and let Dr. Schwartz and your TA know by sending them an email.) Add the following equates ~~defines~~ to your .asm ~~.c~~ file

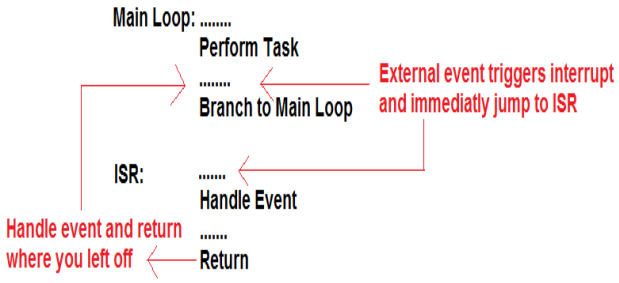
```
.equ CR = 0x0D      ~~#define CR 0x0D~~
.equ LF = 0x0A      ~~#define LF 0x0A~~
.equ ESC = 0x1B     ~~#define ESC 0x1B~~
```

The best way to write and debug code is to test **as you write it**. I suggest that you start by writing a simple program to read a character (using IN_CHAR), then write a simple program to output a character (using OUT_CHAR), and then write program to echo what is received with IN_CHAR by transmitting the read data using OUT_CHAR. Write a final simple program to test your OUT_STRING subroutine before finally writing the required program to display the menu and execute the menu specified tasks.

PART D: Interrupt Driven Receiving

We have already used interrupts in part A of this lab. Now you will use your knowledge to create an interrupt driven echo program.

1. Initialize USART as in part C, except also enable medium level receiver interrupt.
2. Turn on appropriate PMIC interrupt level in the control (PMIC_CTRL) register.
3. Write a receiver ISR that echoes the received data to the transmitter. As demonstrated in class on the examples online, writing an ISR is very similar to writing a subroutine, with the main differences of often needing to clear a flag (caused by the interrupt) and returning with an RETI instead of a RET. You will also generally need to push and pop several registers.
4. To demonstrate the concept of an interrupt driven program, after your initializations, toggle an LED "forever" with a 0.5 second delay in a loop in the main routine, and do nothing else! At the same time, if you type a key on your terminal, your program should echo it back inside an ISR. The interrupt should now do all the work for you. A properly interrupt driven program should have the following format:



PRE-LAB QUESTIONS

1. List the XMEGA's USART registers used in your programs and briefly describe their functions.
2. What is the difference between synchronous and asynchronous communication?
3. What is the difference between serial and parallel communication?
4. List the number of bounces from part A of this lab. How long (in ms) is your delay routine for debouncing?
5. What is the maximum possible baud you can use **for asynchronous communication** if your board runs at 2 MHz? Support your answer with the values you would place in any special registers that are needed.

IN-LAB REQUIREMENTS

1. Part A: Demonstrate your external interrupt program executing on your board.
2. Part C: Demonstrate your USART menu program executing on your board.
3. Part D: Demonstrate your interrupt driven echo program executing on your board.
4. Answer any questions your TA may ask you about your hardware and software.

APPENDIX

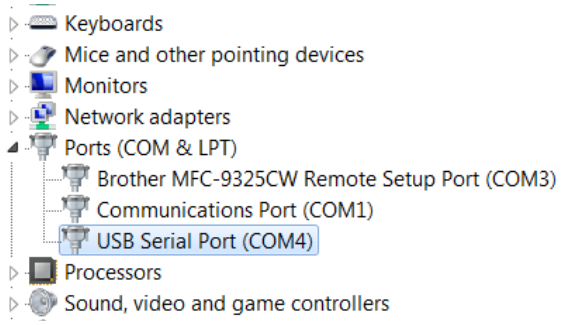
Using Bray Terminal

You will first need to download Bray from the following link: from https://sites.google.com/site/terminalbpp/.

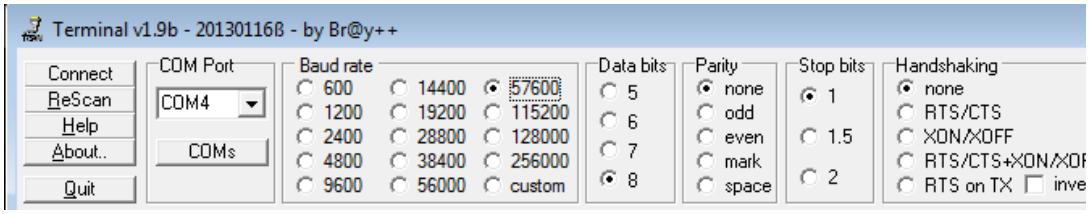
After you write your program, you will need to configure the Br@y++ terminal as follows:

1. With your XMEGA Board powered, and with both USB cables plug in, you will need to determine the virtual serial port that was created. To do this:
    a. Right click **Computer**
    b. Select **Properties**
    c. Click **Device Manager**
    d. Check to see the **Ports (COM & LPT)** associated with the UART:

Lab 5: Interrupts and Serial Communication

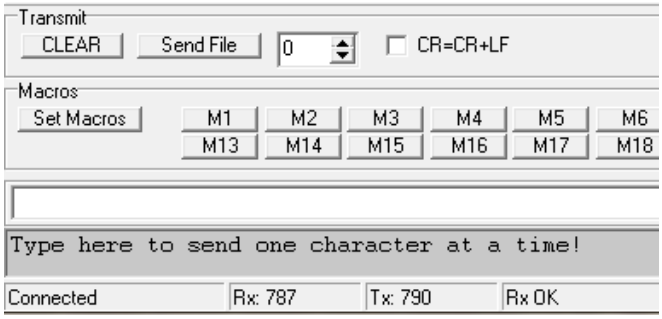


2. Open up Br@y++ terminal and match the settings that you wrote in your program, and the COM Port. Note: Set handshaking (not shown below) to **none**. Note: Select the COM port that you found in the Device Manager. The Atmel board will show up under USB Serial Port. It is COM4 in this screenshot. (Note that the values are not what you need for your lab.)

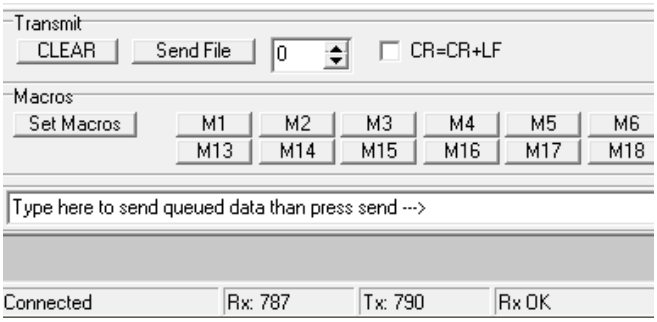


3. Select "Connect" on the top left. If everything is installed correctly, you should **NOT** get a warning, and be ready to transmit and receive data.

4. There are two ways you can transmit data to your XMEGA:
   a. **Single Key Press**: Place your cursor at the bottom of the Br@y++ terminal in the **Transmit** section and begin typing:



   b. **Send Queued Data:** Place your cursor in the white box and type a message; when done press "-> Send' on the right side (not shown below) to send the entire message:



5. When Br@y++ terminal receives data, it will be displayed in the receive section, the data can be viewed in ASCII or HEX. We will use ASCII.