Lab Report 4

Nathan Tipton, Partner: Tarrin Rasmussen

October 24, 2016

Contents

1 Objectives

The purpose of this lab is to become familiar with the PS/2 protocol and impementation of an interrupt service routine.

2 Overview

For this lab, we will be creating a hardware keylogger which will display ASCII characters via RS-232.

2.1 Requirements

1. The ISR must be triggered for each clock tick.
2. The keystrokes must be stored in memory.
3. You need to have a button that starts/stops the keylogger. Once the keylogger has been stopped, the captured keystrokes should be sent to the computer via RS-232.
4. You must make use of interrupts to interface with the button. The button should have a higher priority than the clock interrupt.
5. The program only needs to be able to capture and display lowercase alphanumeric characters and space.

3 Procedure

3.1 Connecting to PC

A PS/2 to USB adapter and PS/2 extension cable will be used to connect the keyboard to the PC. The extension cable will be spliced onto a breadboard to allow a connection with the microcontroller.

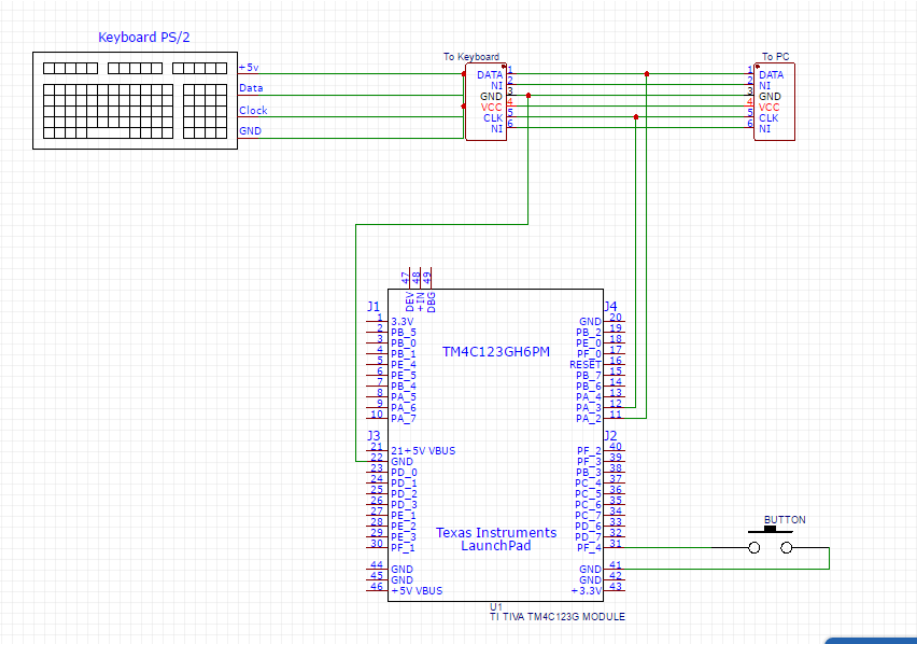

Figure 1: Schematic

3.2 PS/2 protocol

The microcontroller needs to be connected to the data, clock and ground lines of the ps/2 connection. We used the multimeter to determine which pin of the ps/2 connector was with which color wire. We used a schematic of ps/2 to tell us which pins were data and clock. We verified this information by connection the keyboard up to the PC. We then connected the logic analyzer to the breadboard in order to see the signals passed over the lines. This allowed us to see the clock and data signals as seen in figure 2. The clock signal is only sent when a key is pressed. When connecting the microcontroller to the keyboard we had to make sure our pins were 5v tolerant. Pins on the TIVA C which are not 5v tolerant: PD4, PD5, PB0, and PB1. The keyboards output is 5v which would cause problems on these pins.

3.2.1 Configuring Interrupts

1. Set up GPIO for the pins. In this case PA [3:2] are used for the clock and data lines.

2. Configure the GPIO interrupt sense. Setting the pins to detect edges.

3. PA2 is set to be configured to be controlled by the GPIO Interrupt Event (GPIOIEV) register. This is set using the GPIO Interrupt Both Edges

register.

4. In the GPIOIEV register, PA2 is set to detect falling edges.
5. PA2 is set in the GPIO Interrupt Mask to enable an interrupt from the pin to be sent to the interrupt controller.
6. The interrupt is enable in the NVIC Interrupt set enable register.
7. All other interrupts are set up using the same registers.
8. The interrupts are assigned priorities using the Interrupt Priority Registers. Values range from 0 to 7, with the lower value having a higher priority.
9. Interrupts occur with each clock tick as well as when the start/stop button is pressed.

3.2.2 Testing

Using the Logic Analyzer, we verified the packets being sent via PS/2. The clock and data signals were captured from the keyboard as seen in figure 2. The caputred data was converted into ASCII as seen in figure 3. Stored data is displayed on the terminal when the stop button triggers it's interrupt. The displayed message can be seen in figure 4.

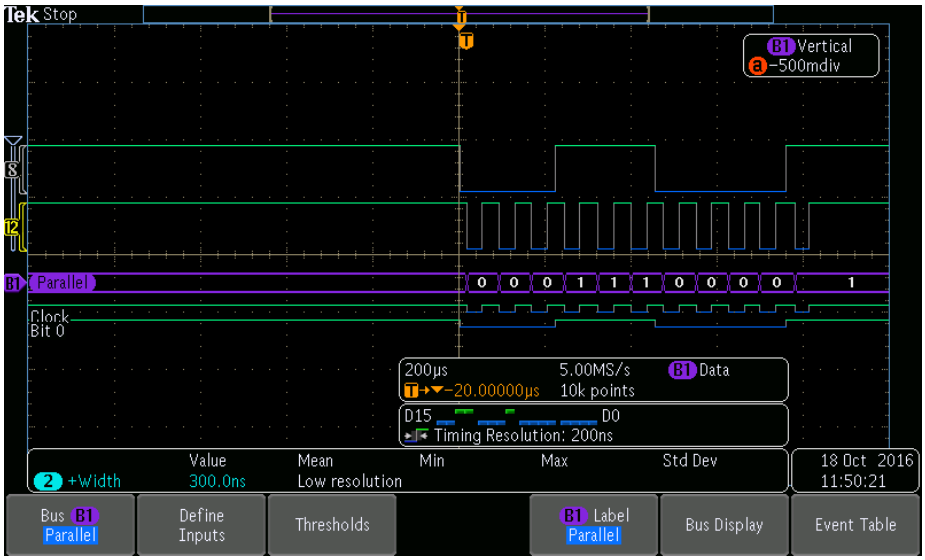

Figure 2: Clock and data signal

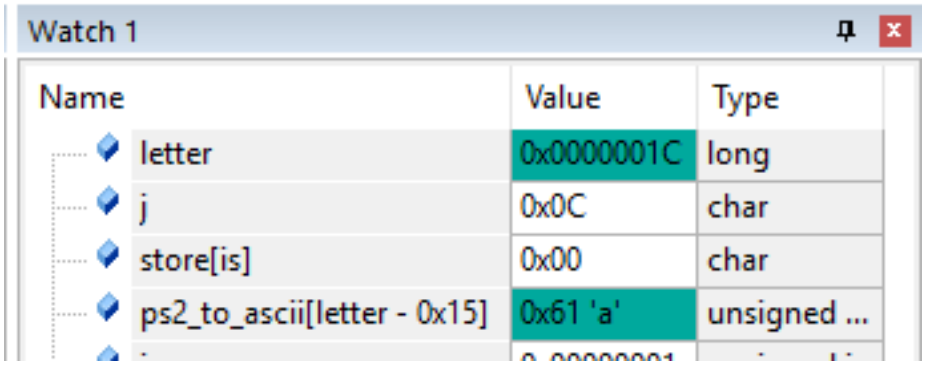

Figure 3: Captured data and conversion

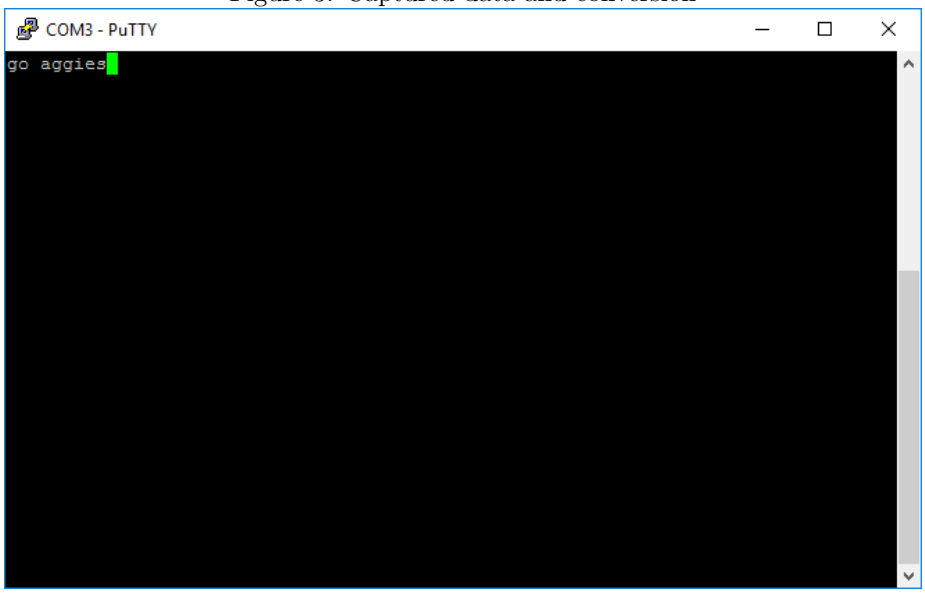

Figure 4: Displayed ASCII on terminal

4 Conclusion

Using C instead of assembly was slight challenging to begin with. We were stuck in an assembly mindset. Configuring the interrupts was challenging due to the lack of straight forward steps. There wasn't a set of steps to follow as in previous labs. This caused us to miss configuration steps initally preventing the interrupts from working correctly. We had the most trouble importing the data from the keyboard correctly. The first character read by the program was correct while proceeding characters were incorrect. We discovered this was caused by incorrect reading of the packet. Each read after the first was shifted causing incorrect values. Adjusting the number of bits read corrected this problem and only the data bits were used. We also ran into the problem of repeated characters. This was caused by the repeat signal upon key release. Having the program look for the 0xF0 signal, which indicated the release and skipping the following character, solved the problem.

5 Appendix

5.0.1 Code

```
#include "TM4C123.h"
#include "ps2_to_ascii.h"
#include <stdbool.h>


bool p[11];
char en = 0;
bool skip = 0;
char i = 0;
char bin[11] = { 0, 1, 2, 4, 8, 16, 32, 64, 128, 0, 0 };
char store[10000];
int *nvic = (int *) 0xE000E100;
int *priority = (int *) 0xE000E400;
unsigned int is = 0;
void GPIOA_Handler(void) {
 char j;
 long letter = 0;

 GPIOA->ICR = 4;

 if (en == 1) {
 p[i] = GPIOA->DATA & 8;
 i++;

 if (i == 11) {
    i = 0;

    for (j = 0; j < 12; j++)
     letter += p[j] * bin[j];

    if (letter == 0xF0) skip = 1;

    else if (skip == 1) skip = 0;

    else {
     store[is] = ps2_to_ascii[letter - 0x15];
```

```
    is++;
   }
  }
 }
}


void GPIOF_Handler(void) {
 int k;
 int l;
 for (l = 0; l < 500000; l++) {}
 GPIOF->ICR = 0x10;
  if (en == 1) {

  for (k = 0; k <= is; k++) {
   while (UART0->FR & 0x20) {}
   UART0->DR = store[k];
  }
  is = 0;
  store[0] = 0;
  en = 0;
 }
  else en = 1;
}
int main(void) {



 //
 //Setup UART Module 0
 //

 //Reset Clock
  SYSCTL->RCC = 0x078E3AD1; //RCC Reset

 //Enable UART Module
 SYSCTL->RCGCUART |= 1; //Enable Module 0

 //Enable GPIO Clock
 SYSCTL->RCGCGPIO |= 0x21; //Enable Port A,F

 //Enable ALT Function
 GPIOA->AFSEL |= 3; //Set GPIO Port A Pins 0, 1

 //Digital Enable
 GPIOA->DEN |= 3;

 //Disable UART
 UART0->CTL &= 0xFFFFFFFE;

 //Load INT portion of BRD
 UART0->IBRD = 104;

 //Load FRAC portion of BRD
 UART0->FBRD = 11;

 //Set Line Control
 UART0->LCRH = 0x72; //Parity, FIFOs, and 8-bit length

 //Set Baud Clock Source
 UART0->CC = 5; //PIOSC

 //Enable UART
 UART0->CTL |= 1;


 //
 //Setup GPIO Port A Pins 2,3
```

```
//

//Enable GPIO Clock
//Done above

//Set Pin Direction
GPIOA->DIR &= 0xFFFFFFF3; //Pins 2,3 input

//Disable ALT Function
GPIOA->AFSEL &= 0xFFFFFFF3; //Pins 2,3 GPIO

//Digital Enable
GPIOA->DEN |= 0xC; //Pins 2,3 enable

//Set Interrupt Sense
GPIOA->IS &= 0xFFFFFFFB; //Pin 2 edge

//Set Interrupt Both Edges
GPIOA->IBE &= 0xFFFFFFFB; //Pin 2 controlled by IEV

//Set Interrupt Event
GPIOA->IEV &= 0xFFFFFFFB; //Pin 2 falling edge

//Set Interrupt Mask
GPIOA->IM |= 0x4; //Pin 2


//
//Setup GPIO Port F Pin 4
//

//Enable GPIO Clock
//Done above

//Unlock Port
GPIOF->LOCK = 0x4C4F434B;

//Unlock All Pins
GPIOF->CR = 0xFF;

//Set Pin Direction
GPIOF->DIR &= 0xFFFFFFEF; //Pin 4 input

//Disable ALT Function
GPIOF->AFSEL &= 0xFFFFFFEF; //Pin 4 GPIO

//Set Drive Strength
GPIOF->DR8R |= 0x10; //Pin 4 8mA

//Set Pin Function
GPIOF->PUR |= 0x10; //Pin 4 pull up

//Digital Enable
GPIOF->DEN |= 0x10; //Pin 4 enable

//Set Interrupt Sense
GPIOF->IS &= 0xFFFFFFEF; //Pin 4 edge

//Set Interrupt Both Edges
GPIOF->IBE &= 0xFFFFFFEF; //Pin 4 controlled by IEV

//Set Interrupt Event
GPIOF->IEV &= 0xFFFFFFEF; //Pin 4 falling edge

//Set Interrupt Mask
GPIOF->IM |= 0x10; //Pin 2

//Enable NVIC????
```

```
 nvic[0] = 0x40000001;
 priority[0] |= 0x20;
 while(1) {}

}
```