Introduction

This document describes the design for a hardware keylogger prototype. A hardware keylogger intercepts all the keystrokes that a user types to a keyboard and stores the keystrokes in memory for later retrieval. Practical application of this technology is frequently utilized by military, government, and law enforcement agencies. Most often a keylogger is used for covert recording of user passwords.

Overview

This document discusses, in general, the electrical and software design for the hardware keylogger prototype. It includes the requirements, design details and commentary. A hardware diagram, logic analyzer output, and code debugging screenshots are included to reinforce the results and conclusions of the lab. The complete code is located in the Appendix.

Requirements

The following are the given requirements for the hardware key logger prototype:

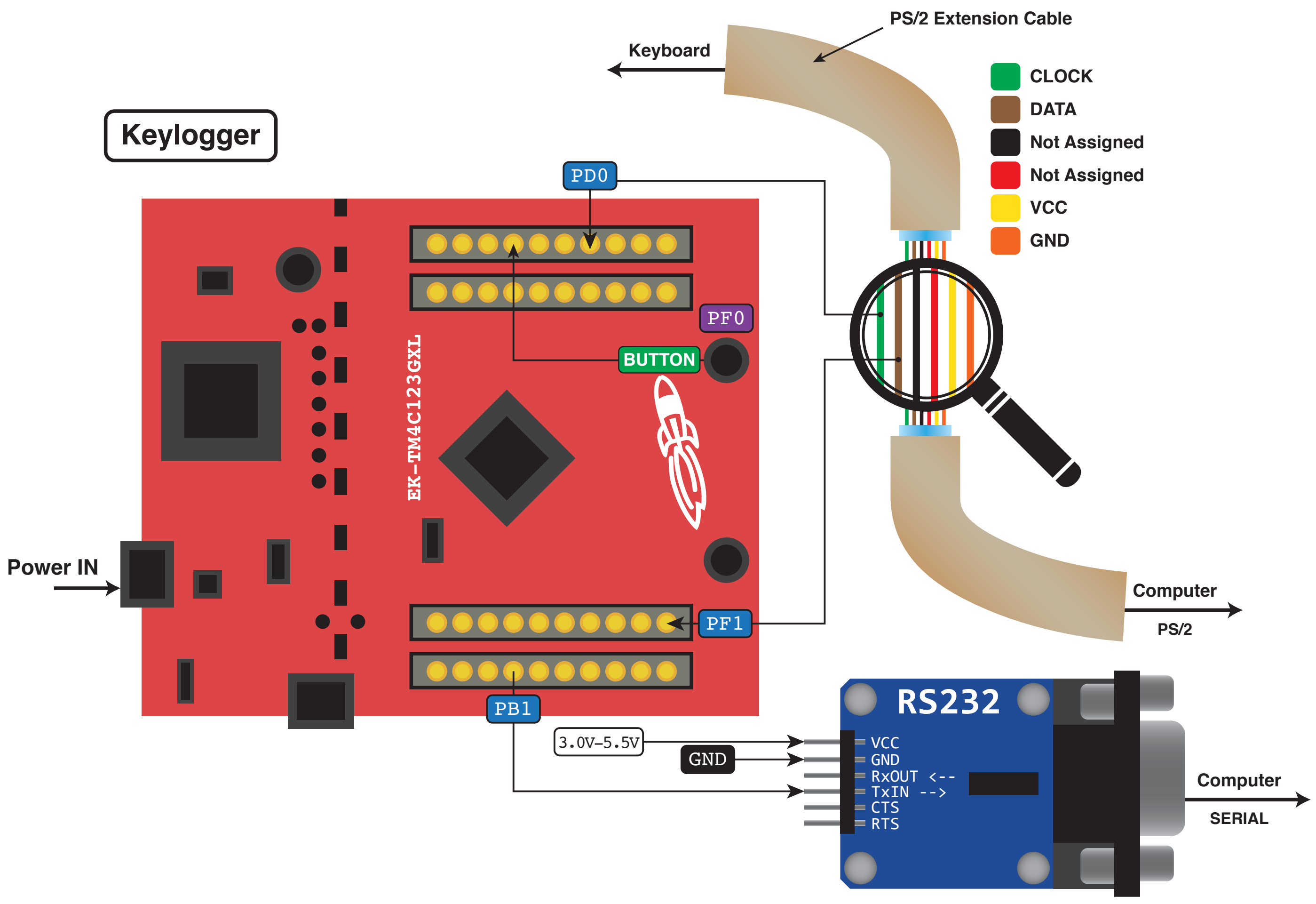
1. The ISR must be triggered for each clock tick.
2. The keystrokes must be stored in memory.
3. You need to have a button that starts/stops the keylogger. Once the keylogger has been stopped, the captured keystrokes should be sent to the computer via RS-232.
4. You must make use of interrupts to interface 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.

Precaution

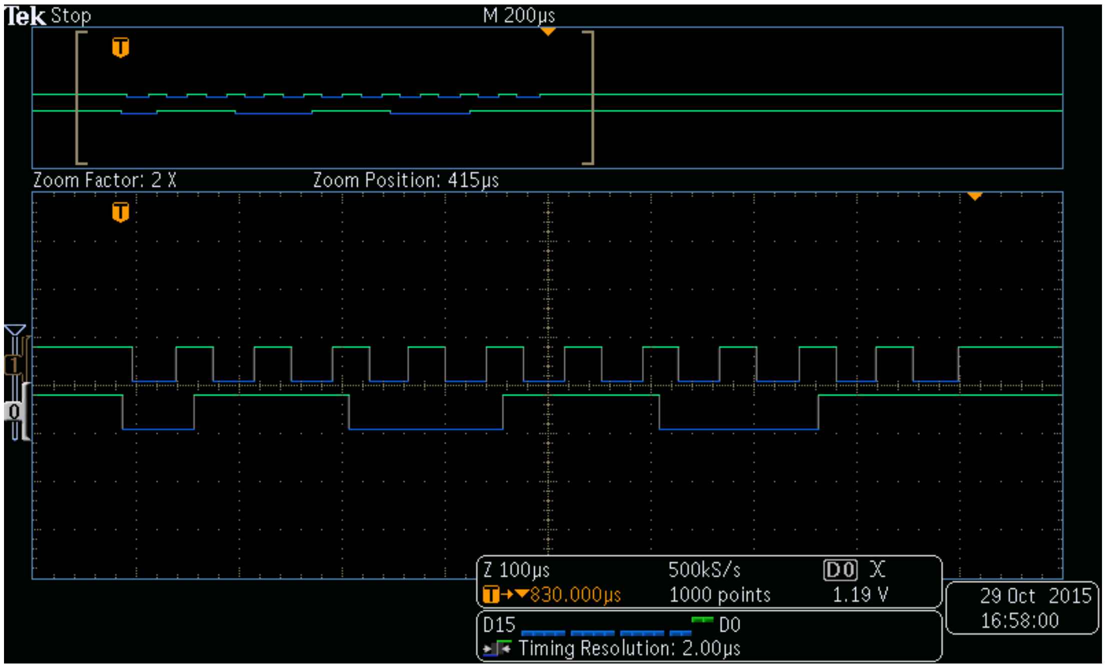
The keyboard utilizes 5V logic while the Tiva C uses 3.3V logic. If connected improperly, this can fry the Tiva C microcontroller. According to the Tiva C Data Sheet (MDS), “All GPIO signals are 5-V tolerant when configured as inputs except for **PD4**, **PD5**, **PB0** and **PB1**, which are limited to 3.6 V” (pg. 647 – MDS).

Design Details

Hardware Diagram



Logic Analyzer Output



● The clock signal of the keyboard with data from the keyboard.

Commentary

Testing

Behavior of the code has been exhaustively observed via the debugger mode of Keil uVision. The pressing of keyboard keys shows in real-time the byte to ASCII translation in the memory space selected on the microcontroller for storage. Additionally the reading and transferring of those ASCII bytes to the TX register for transmission to the PC was also observed and deemed correct behavior. See Appendix at end of program code for screenshot.

Conclusions

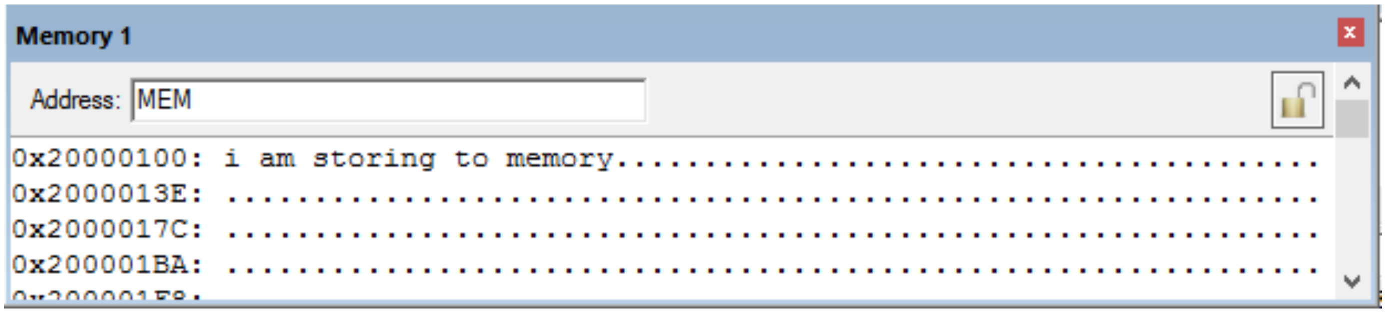
Lab 4 demonstrates the function of interrupts on the Tiva C microcontroller and the UART module such that the microcontroller is able to interface with the RS-232 to transmit ASCII characters byte-by-byte. The interrupts were configured to microcontroller's onboard button SW2 and the clock signal generated by the keyboard. Although the software behavior exhaustively demonstrates that the ASCII characters are being stored into the TX register for transfer to the PC, the computer terminal was not successfully able to receive the ASCII bytes and display the encoded messages on screen. In the program written to the microcontroller, input from the onboard button successfully turns the keylogging functionality on and off. Likewise, the logic analyzer shows the clock signal of the keyboard and the associated data of the keystrokes recorded in the microcontroller's memory.

Appendix

Program Code

```
// UART TX: 9600 baud w/16 MHz clock

// system control base addr
unsigned char *SYSCTL = (unsigned char *) 0x400FE000;
//UART1 shares pins with Port A (RX: PA0, TX: PA1)
unsigned char *PB = (unsigned char *) 0x40005000;
unsigned char *PF = (unsigned char *) 0x40025000;
unsigned char *PD = (unsigned char *) 0x40007000;
unsigned char *MEM = (unsigned char *) 0x20000100;

volatile unsigned char c = 0;
volatile unsigned char valuecounter;
volatile unsigned int value = 0;
volatile unsigned char *i=(unsigned char*) 0x20002000;
volatile unsigned char *k=(unsigned char*) 0x20002500;

unsigned char ps2_to_ascii[] = {
       'q','1',0x00,0x00,0x00,'z','s','a','w','2',0x00,

       0x00,'c','x','d','e','4','3',0x00,0x00,' ','v','f','t','r','5',0x00,0x00,

       'n','b','h','g','y','6',0x00,0x00,0x00,'m','j','u','7','8',0x00,

       0x00,',','k','i','o','0','9',0x00,0x00,'.',0x00,'l',0x00,'p'};

// UART1 base
unsigned char *UART1 = (unsigned char *) 0x4000D000;
unsigned char UART1_D __attribute__((at(0x4000D000)));
volatile int UART1_R __attribute__((at(0x4000D000)));
volatile unsigned int UART1_STAT __attribute__((at(0x4000D018)));
unsigned char *SYSNVIC = (unsigned char *) 0xE000E000;

//set-up of GPIO Port F
volatile int PF_DATA_R __attribute__((at(0x400253FC)));
volatile int PF_DIR_R __attribute__((at(0x40025400)));
volatile int PF_AF_R __attribute__((at(0x40025420)));
volatile int PF_DEN_R __attribute__((at(0x4002551C)));
volatile int PF_ULK_R __attribute__((at(0x40025520)));
volatile int PF_CR_R __attribute__((at(0x40025524)));
volatile int PF_PUL_R __attribute__((at(0x40025510)));

//set-up of GPIO Port D
volatile int PD_DATA_R __attribute__((at(0x400073FC)));
volatile int PD_DIR_R __attribute__((at(0x40007400)));
volatile int PD_AF_R __attribute__((at(0x40007420)));
volatile int PD_DEN_R __attribute__((at(0x4000751C)));
volatile int PD_ULK_R __attribute__((at(0x40007520)));
volatile int PD_CR_R __attribute__((at(0x40007524)));
volatile int PD_PUL_R __attribute__((at(0x40007510)));

//set-up of Interupt Port F
volatile int PF_IS_R __attribute__((at(0x40025404)));
volatile int PF_IEV_R __attribute__((at(0x4002540C)));
volatile int PF_IBE_R __attribute__((at(0x40025408)));
volatile int PF_IM_R __attribute__((at(0x40025410)));
volatile int PF_ICR_R __attribute__((at(0x4002541C)));

//set-up of Interupt Port D
volatile int PD_IS_R __attribute__((at(0x40007404)));
volatile int PD_IEV_R __attribute__((at(0x4000740C)));
volatile int PD_IBE_R __attribute__((at(0x40007408)));
volatile int PD_IM_R __attribute__((at(0x40007410)));
volatile int PD_ICR_R __attribute__((at(0x4000741C)));

//set-up of GPIO PORT B
volatile int PB_DATA_R __attribute__((at(0x400053FC)));
volatile int PB_DIR_R __attribute__((at(0x40005400)));
volatile int PB_AF_R __attribute__((at(0x40005420)));
volatile int PB_DEN_R __attribute__((at(0x4000551C)));
volatile int PB_ULK_R __attribute__((at(0x40005520)));
volatile int PB_CR_R __attribute__((at(0x40005524)));
volatile int PB_PUL_R __attribute__((at(0x40005510)));

void GPIOInit()
{
  SYSCTL[0x608] = 0x2A; // Clock Enable

  // >> GPIO Port F (PF)
  // 1. Unlock Pins / Disable Alt. Function
  PF_ULK_R=0x4C4F434B;
  // Unlock Code (pg. 681 - MDS)
  PF_CR_R=0x03;
  // 2. Set Pins as Input or Output
  PF_DIR_R=0xFC;
  // 3. Set for Pull-Up or Pull-Down
  PF_PUL_R=0x03;
  // 4. Enable Pins
  PF_DEN_R=0x03;

  // >> GPIO Port D (PD)
  // 1. Unlock Pins / Disable Alt. Function
  PD_ULK_R=0x4C4F434B;
  PD_CR_R=0x03;
  // 2. Set Pins as Input or Output
  PD_DIR_R=0xFE;
  // 3. Set for Pull-Up or Pull-Down
  PD_PUL_R=0x01;
  // 4. Enable Pins
  PD_DEN_R=0x01;
}
void UART1Init()
{
  // 1. enable clock: uart then port
  SYSCTL[0x618] = 0x2; //UART1

  // 2. PA1: enable alt. func. and pin
  PB_ULK_R = 0x4C4F434B;

  PB[0x420] = 0x3; //AFSEL
  PB[0x51C] = 0x3; //DEN

  // 3. disable UART1
  UART1[0x30] = 0x0;
  // 4. set baudrate divisor
  // BRD = 16e6/(16*9600)=104.1667
  // integer portion: int(104.1667)=104=0x68
  UART1[0x24] = 0x68;
  //  fractional portion: int(0.1667*2^6+0.5)=11=0xB
  UART1[0x28] = 0xB;

  // 5. set serial parameters: 8-bit word, start/stop/parity bits
  UART1[0x2C] = 0x62;

  // 6. enable tx and uart
  UART1[0x30] = 0x01;
  UART1[0x31] = 0x01;
  SYSNVIC[0x100]=0x08; //0x40000008 set from little indian.
  SYSNVIC[0x101]=0x00; //This sets interupts on PORTD and on PORTF.
  SYSNVIC[0x102]=0x00;
  SYSNVIC[0x103]=0x40;
}
void INTRInit()
{
	//setting up interupt for PORT F0
	PF_IS_R=0x0;//set to trigger on edge 0=edge 1=level
	PF_IEV_R=0x0;//setting edge to 1 which equals rising edge 0=falling 1=rising
	PF_IBE_R=0x0;//set to 0 which takes the input from the IEV0=from IEV 1=both rising and falling
edge
	PF_IM_R=0x1;//set to 1 for it to recognize as an interupt 0=masked/not seen 1=enabled for
interupt

	//setting up interupt for PORT D0
	PD_IS_R=0x0;//set to trigger on edge 0=edge 1=level
	PD_IEV_R=0x0;//setting edge to 1 which equals rising edge 0=falling 1=rising
	PD_IBE_R=0x0;//set to 0 which takes the input from the IEV0=from IEV 1=both rising and falling
edge
	PD_IM_R=0x1;//set to 1 for it to recognize as an interupt 0=masked/not seen 1=enabled for interupt

}
void GPIOD_Handler(void)//keyboard clock
{
	unsigned char value;
	int pointer = *i;
	int count = *k;
	if(pointer!=0 && pointer<9)
	{
		value = PF_DATA_R;
		value &= 0x2;
		value = value >> 1;
		value = value << (pointer - 1);
		MEM[count] |= value;

		if(pointer == 8)
			{
				MEM[count] = MEM[count] - 0x15;
				MEM[count] = ps2_to_ascii[MEM[count]];
				count ++;
				*k = count;
			}
	}
	pointer++;
	*i = pointer;
```

```
        PD_ICR_R=0x1;//1=interupt recognizes that it has been taken care of. 0=nothing happens
}
void GPIOF_Handler(void)//button
{
        unsigned int d;
        int count = *k;
        for(d = 0; d < count; d++)
        {
                UART1[0x00] = MEM[d];
                while(UART1_STAT == 0x80);//
                MEM[d]=0x0;
        }
        *k = 0;
        PF_ICR_R=0x1;//1=interrupt recognizes that it has been taken care of. 0=nothing happens
}
int main(void)
{
// This is a look up table for basic key strokes.
// This translates from ps/2 to ascii.
// The table starts at 0x15, so you should shift
// the table by 0x15 when you reference it.

  *i = 0;
  *k = 0;

        GPIOInit();
        UART1Init();
        *UART1 = 0x0D;
        *UART1 = 0x0A;
        INTRInit();
```


Memory 1

Address: MEM

```
0x20000100: i am storing to memory..............................................
0x2000013E: ..............................................................
0x2000017C: ..............................................................
0x200001BA: ..............................................................
```