Report 3: Communication Systems & Microcontroller on FPGA

Ibrahima Diallo, Yiwei Ling, Aaron Marsch

Syracuse University

Introduction

The purpose of experiment 8 and 9 was to introduce our team to data transmission. In experiment 8 we learned how to implement i2c communication to a PIC16F1769 microchip by using Analog discovery to scope the data. This helped us create the PIC16F1769 as the secondary device, with logic to create a specific output for the input received from the i2c bus. Experiment 9 contrasted from Experiment 8, where instead we used UART and NIOS 2 to create a custom microcontroller. In experiment 9 we had to use our prior experience from CSE262 where we learned about Quartus and the fundamentals of FPGAs. We then proceeded to program an FPGA to run Hello World and make a switch controlling LED GPIO. Our main questions for experiments 8 and 9 were, “How would we use a logic analyzer to send bytes of data?” and “How will we write a programming interface to an FPGA? ”.

Experiment 8

Introduction

The goal of this experiment is to introduce the i2c communication protocol. I2c stands for Inter-Integrated Circuit. Invented in 1982, i2c allows multiple peripheral chips to communicate with one or more controller chips. In i2c the serial clock (SCL) carries the clock signal and the serial data line carries the data. The serial data line is split up into a start condition, address frame, read/write bit, acknowledge/no acknowledge bit, and a stop condition. I2c uses synchronous communication. Pull up resistors are necessary because the communication signals because they protect the gates from shorts. More on the i2c communication protocol can be found here. In the first part of this experiment we will set up and run the PIC microcontroller as the main device writing to an address and in the second part we will set up the PIC as the secondary device reading from Analog Discovery.

Materials

- PIC16F1769
- Analog Discovery 2
- Snap Programmer
- 2 1kΩ resistors
- Wires

Experimental Setup

The code for the first half of the experiment can be seen in Figure 8.1. In the configuration bits set PLLEN = ON to permanently enable the PLL. Make OSCCON = 0x70 to set the clock frequency. Create the i2c_main_init(void) function as shown in Figure 8.1. SSP1CON1 register is configured so that the overflow bit is not set since each reception (and transmission) is initiated. The Fosc clock frequency is 25kHz and the i2c clock frequency is 100 kHz.

Write out the i2c_main_wait(), i2c_main_start(), i2c_main_stop(), and i2c_main_write_data() data methods from Figure 8.1.

Wire the Snap programmer to the PIC microcontroller by connecting Snap's pin 1 (Vpp) to the PIC's pin 4 (Vpp), pin 3 (GND) to pin 17 (Vss), pin 4 (PGD) to pin 19 (ICSPDAT), and pin 5 (PGC) to pin 18 (ICSPCLK). Wire pin 16 (RC0 on the PIC) to DIO0 and RC1 to DIO1. The pins layout for the PIC can be found in Figure 8.2 and the complete circuit can be found in Figure 8.5:. These connect the clock and data lines to the Analog Discovery so that we may be able to see them on Logic.

Experimental Procedure

In Waveforms > Logic > i2c Analyzer > I2C > Spy, set up the i2c Analyzer with DIO0 as the SCL line and DIO1 as the SDA line. The uncommented code in Figure 8.1's main() function will write values 0 to 255 on repeat to an address of 47.

The next step of the experiment is to use Analog Discovery's protocol analyzer to "spy" on the i2c bus from the PIC. Set the protocol analyzer to the right address and you should see output similar to what is shown in Figure 8.3.

For the second half of the experiment, we will be writing from Analog Discovery and the PIC will be the secondary device. Switch from "Spy" to "Master" in Protcol's I2C tab.

Send 1+ (Channel 1) of the Analog Discovery to scope the i2c clock and send 2+ (Channel 2) to scope the i2c data.

Write the i2c_secondary_init(void) and i2c_secondary_poll(p1, p2) methods shown in Figure 8.4. Call i2c_secondary_init(void) once in the main. i2c_secondary _poll uses polling (instead of interrupts) to check if the main device communicated with the PIC. It handles the logic that deals with what to do when a byte is received on the i2c bus. Write the main code from Figure 8.4 that handles the polling and calls the poll function.

Results

After following the steps of this experiment you should have more insight into how i2c works from both the perspective of the primary device and the secondary device. Your results from the first half should resemble Figures 8.6 and 8.3. You can see that the SCL line does resemble a clock and can match the different parts of the SDA line (start bit, stop bit, address frame, etc.) to the guide that Waveforms provides.

The results of the second half of the experiment should resemble Figures 8.7 and 8.8. On the clock line in Figure 8.7 you can see how the PIC pulls down the clock signal on the acknowledge bits of the data signal. This is just one way you can verify that everything is working correctly. In Figure 8.8 we see output with different numbers of bytes being written. Nobody was able to solve why the NAK bit was not working correctly but everything else works fine so we would recommend that you ignore it like our instructor allowed us to.

Experiment 9

Introduction

In this experiment, we will be learning UART and NIOS 2 via Quartus under a developing environment named Eclipse 2. UART is a protocol, stands for universal asynchronous receiver-transmitter.Which sends and receives data format at a configurable transmission speed. We will be implementing UART on a FPGA name DE10 lite. We will be coding and doing the FPGA design and pin planning on Intel's Quartus. The coding part will be under Eclipse which basically using C language.

Materials

- DE10-Lite
- Key pad
- Resistors
- Analog discovery(not necessary)

Experimental Setup

The setup of this project takes up most of the time. First we open Quartus, and start a new project named lab9. Then we selected the correct microchip for our project which is Altera MAX 10 10M50DAF4848C7G. At this part of the lab we do not need to connect anything to the computer.

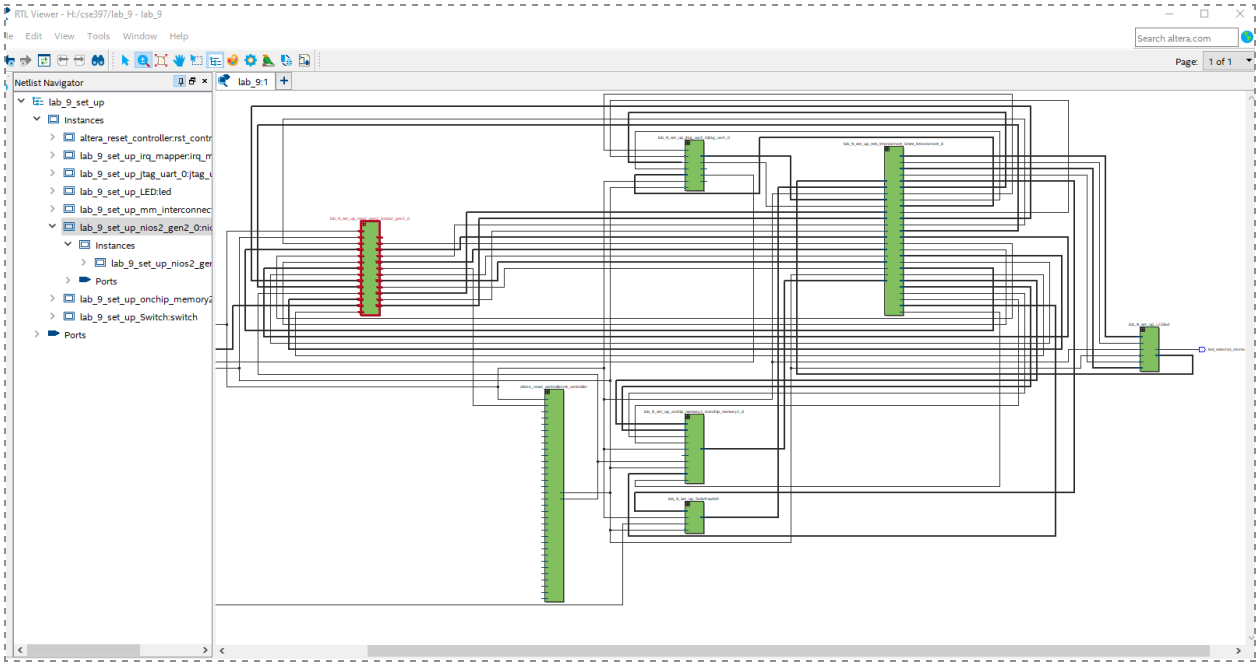
First we need to open the platform designer which allows us to visually see the components for the chip design. The first thing we need is to add a NIOS II processor. We selected the economic version because it suits our DE-10lite board better and uses less hardware resources. At this point, quartus tells us there are many errors, but we will eliminate them when we are done with all the setups.

Next, memory is added to the system. We chose ROM and RAM, also called on-chip memory for our memory. We changed the value from 4000 bytes to 40960 bytes because our FPGA board has more memory than 4k bytes.

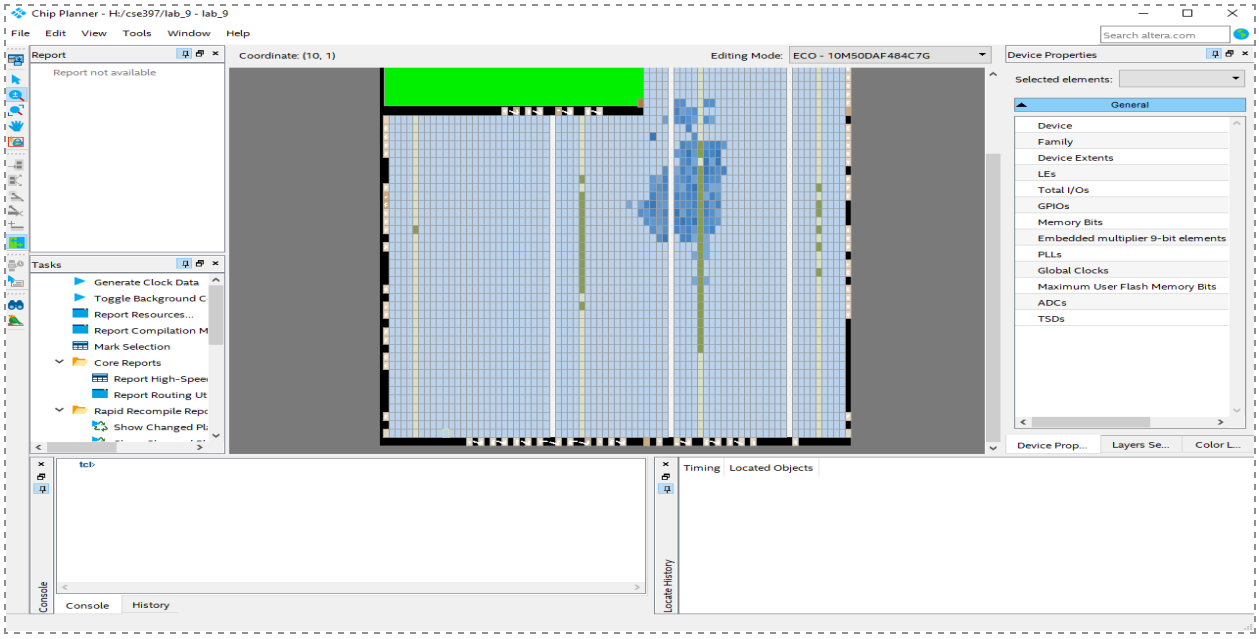
Then, we needed to add a communication protocol JTAG UART Intel FPGA IP , so that we could write the C code to control our microchip.

Then we add general input output pins to the design. We only set one pin as output and one pin as input. The input will be the switch and the output will be the LED so we renamed the pins as their function applies.

After we connected all the components, and fixed the errors, we generated the HDL file and VHDL for our microcontroller, and we could do the pin planning after a full compile. The picture belows shows the RTL view

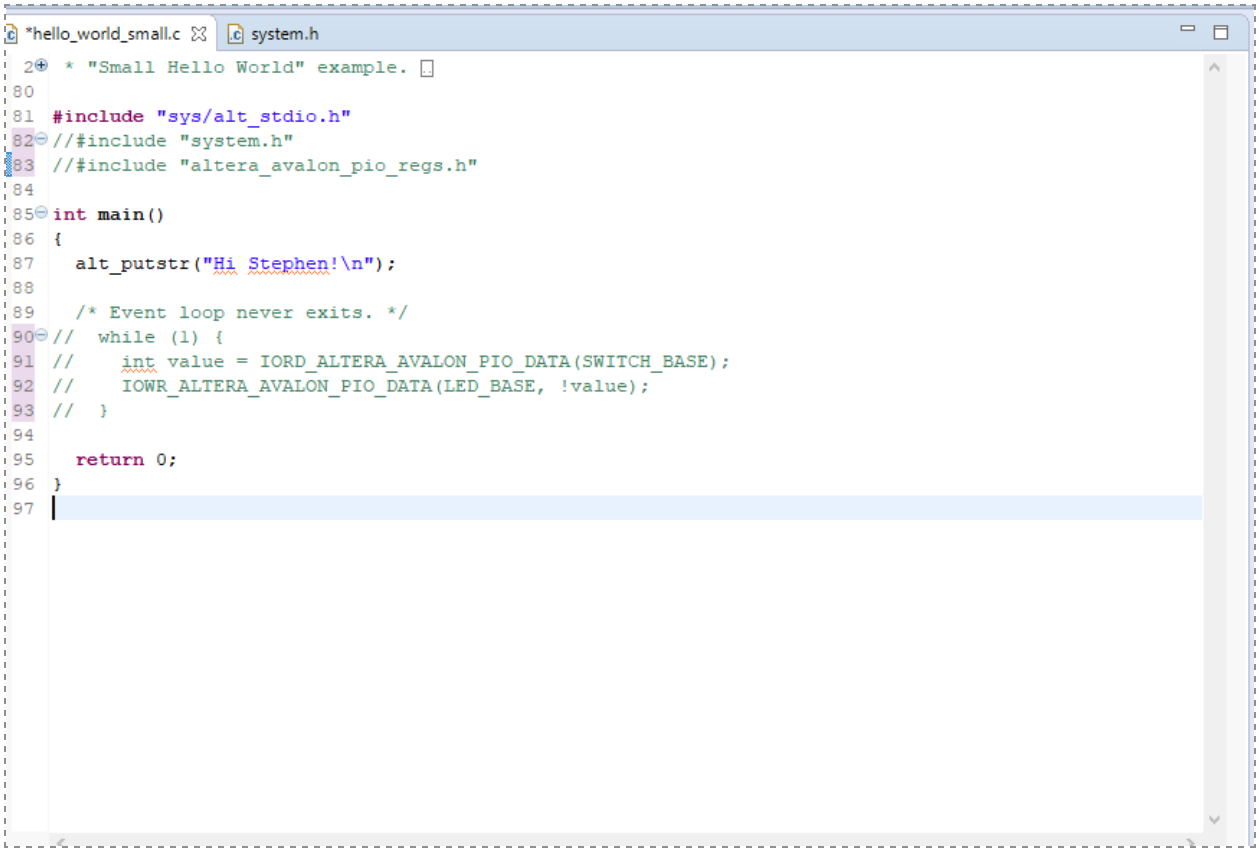


This is the pin chip planner view



Experimental Procedure

After full compiling, we begin to write our first FPGA program using Eclipse. We first have to select a USB-Blaster for it in the Quartus. Then we opened Eclipse, and selected a Hello World sample script. However, when we were in the run configurations in the Eclipse, we had to check the ignored mismatched system ID and the mismatched system time step for it to run. 'Or the system will show Connected system ID hash not found on target at expected base address' error. Then the program successfully runs, and we change the text in the command to see if the code can print something else. Here is the code we use for this signature.



The commented area (green text) is actually for the next signature).

For the second signature, we are using a program to enable a push button on our DE10-lite board. We have to include system.h and altera_avalon_pio)regs.h for the LED. A while loop is needed and inside the while loop, we named our input in this case the switch, and then copy it to the output, the LED, here is the code below.

The last signature we could get to is the control through a key pad, we connect 4 resistors to 4 of the pins of the key pads as pull up resistors as the figure below shows.
The data sheet of DE10- Lite helps us connect the board to the keypad, the figure below is the datasheet.

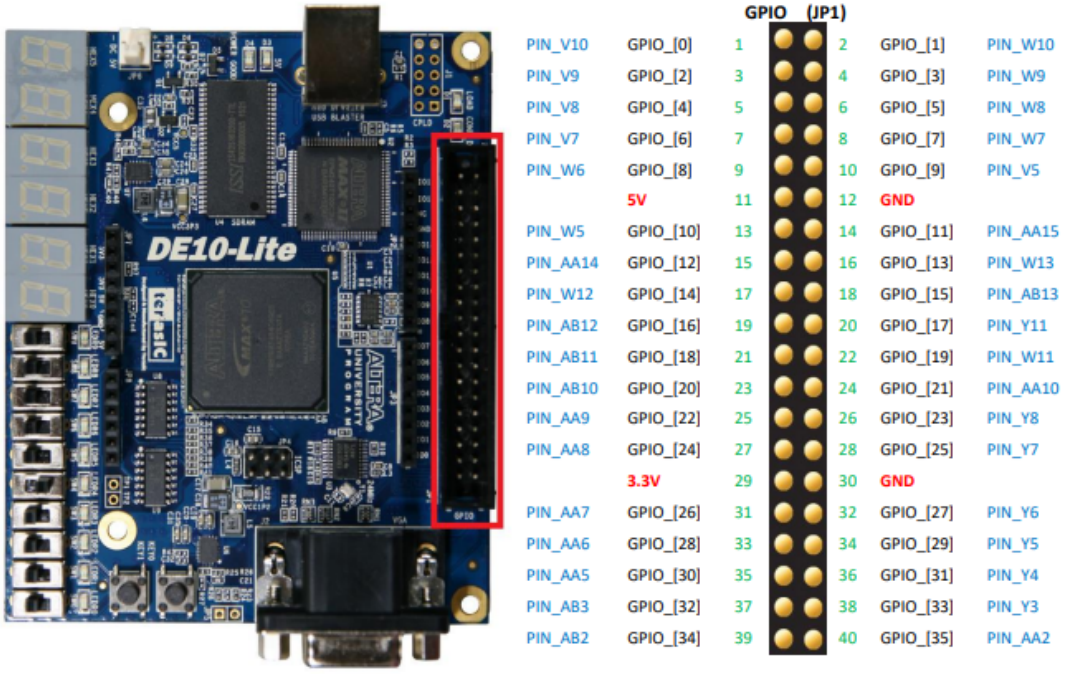
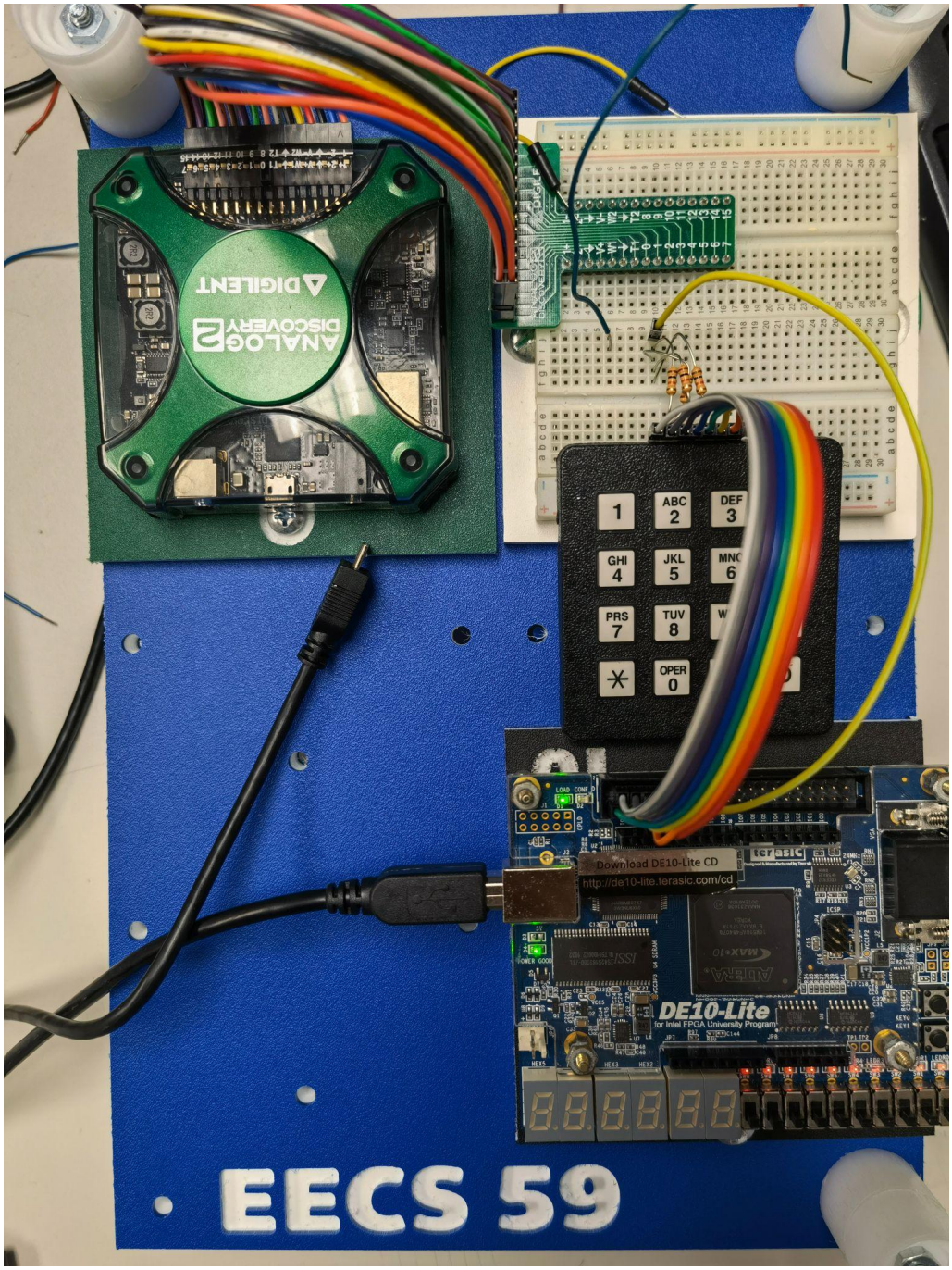


Figure 3-18 I/O distribution of the expansion headers

We selected gpio Pin_V10-V7 and Pin_W6-W10, and V5, and we connected a 5V voltages source for the keypad. Then we connected 4 resistors to 4 of the pins of the key pads as pull up resistors as the figure below shows.



After we configure the GPIO pins in the pin planner, we begin to develop the code for the control of the keypad. Here is our code below:

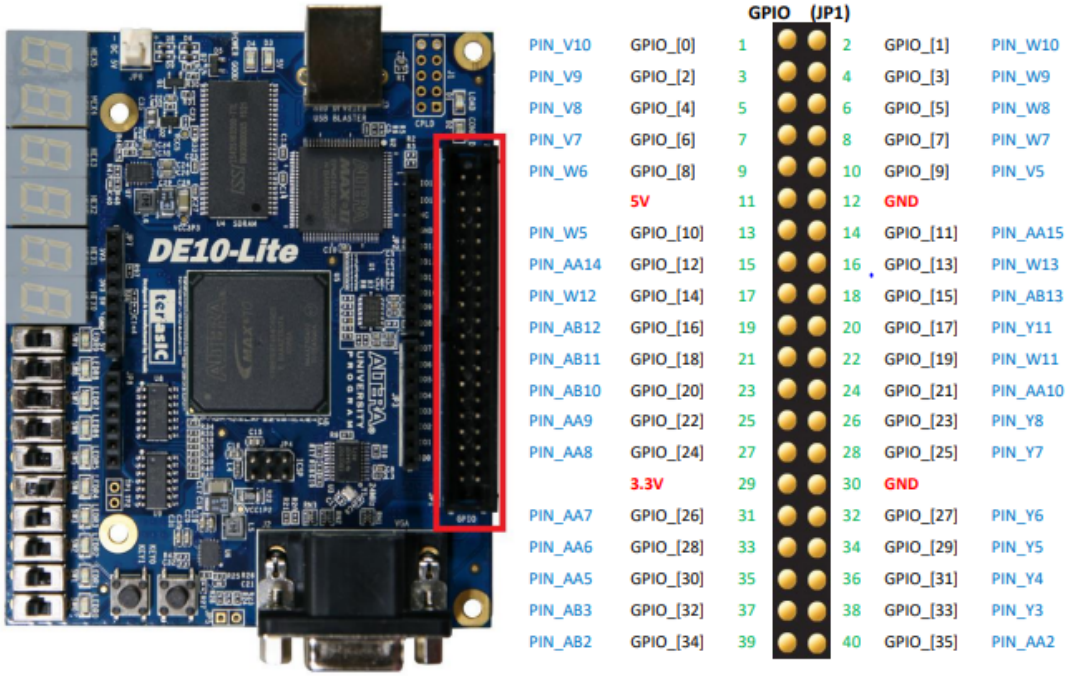


Figure 3-18 I/O distribution of the expansion headers

First we need to connect the keypad to the DE10-LITE , We selected PIN_V10, PIN_V9 ,PIN_V8 , PIN_V7 for the keypad and connected 4 1k resistance between each of them, and we need a 5v voltage source to power the keypad.

```
#include "sys/alt_stdio.h"
#include "stdio.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"
int main()
{
//alt_putstr("Hello World");
/* Event loop never exits. */
while (1){
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b1000);
 int value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 switch(value){
 case 16:
 printf("1\n");
 break;
 case 8:
```

```
 printf("4\n");
 break;
 case 4:
 printf("7\n");
 break;
 case 2:
 printf("*\n");
 break;
}
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0100);
 value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 switch(value){
 case 16:
  printf("2\n");
  break;
  case 8:
  printf("5\n");
  break;
  case 4:
  printf("8\n");
  break;
  case 2:
  printf("0\n");
  break;
 }
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0010);
 value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 switch(value){
  case 16:
    printf("3\n");
    break;
    case 8:
    printf("6\n");
    break;
    case 4:
    printf("9\n");
    break;
    case 2:
    printf("#\n");
    break;
  }
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0001);
 value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 switch(value){
```

```
  case 16:
    printf("A\n");
    break;
    case 8:
    printf("B\n");
    break;
    case 4:
    printf("C\n");
    break;
    case 2:
    printf("D\n");
    break;
  }
}
return 0;
}
```

****

```
int main()
{
//alt_putstr("Hello World");
/* Event loop never exits. */
int f=0;
int f1 = 0;
int f2 = 0;
int f3 = 0;
while (1){
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b1000);
 int value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 if (value == 0){
 f=0;
 }
 switch(value){
 case 8:
 if (f==0){
 printf("D\n");
 f=1;
 }
 break;
 case 4:
 if (f==0){
 printf("C\n");
  f=1;
```

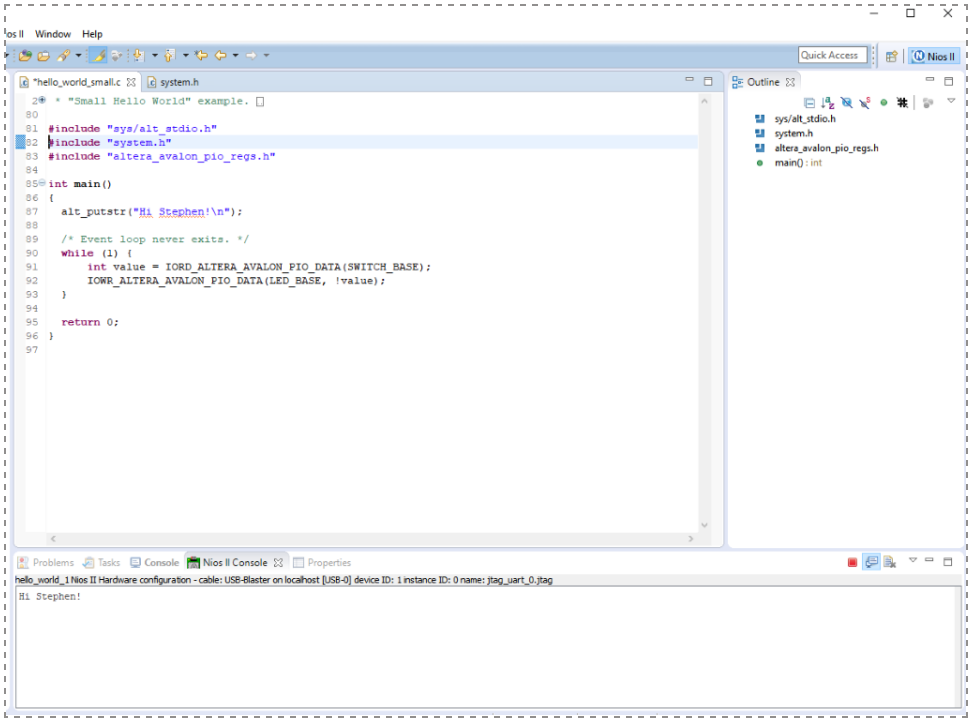
```
 }
 break;
 case 2:
 if (f==0){
 printf("B\n");
  f=1;
 }
 break;
 case 1:
 if (f==0){
 printf("A\n");
  f=1;
 }
  break;

}
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0100);
 value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 if (value == 0){
  f1=0;
  }
 switch(value){
  case 8:
 if (f1==0){
 printf("#\n");
  f1=1;
 }
 break;
  case 4:
 if (f1==0){
 printf("9\n");
  f1=1;
 }
 break;
  case 2:
 if (f1==0){
 printf("6\n");
  f1=1;
 }
 break;
  case 1:
  if (f1==0){
printf("3\n");
 f1=1;
```

```
}
 break;
 }
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0010);
 value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 if (value == 0){
  f2=0;
  }
 switch(value){
    case 8:
   if (f2==0){
        printf("O\n");
     f2=1;
   }
   break;
    case 4:
     if (f2==0){
       printf("8\n");
   f2=1;
    }
    break;
    case 2:
   if (f2==0){
      printf("5\n");
   f2=1;
   }
   break;
    case 1:
  if (f2==0){
     printf("2\n");
     f2=1;
  }
   break;
  }
 IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0001);
 value = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
 if (value == 0){
  f3=0;
  }
 switch(value){
    case 8:
 if (f3==0){
printf("*\n");
 f3=1;
```

```
}
 break;
    case 4:
  if (f3==0){
printf("7\n");
  f3=1;
}
 break;
    case 2:
  if (f3==0){
printf("4\n");
  f3=1;
}
 break;
   case 1:
  if (f3==0){
printf("1\n");
  f3=1;
}
 break;
 }
 for (int i = 0; i < 5000; i++){
 }

}
return 0;
}
```

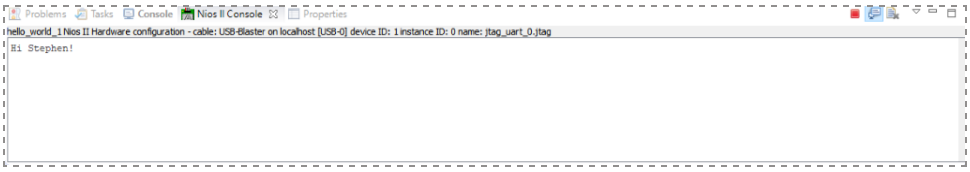
We knew that there are 4 columns and 4 rows, and each 'IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0b0001' corresponds to one column so there will be 4 different cases for the 4 bid binary input. And we need to add another variable f for each case to check if the bottom of the keypad is being pressed or not otherwise the program will have difficulties detecting how many times we have pressed the keypad and It will output something at random instead of one character at a time.

We could not get the correct output with this code, but it outputs something when we press the keypad .

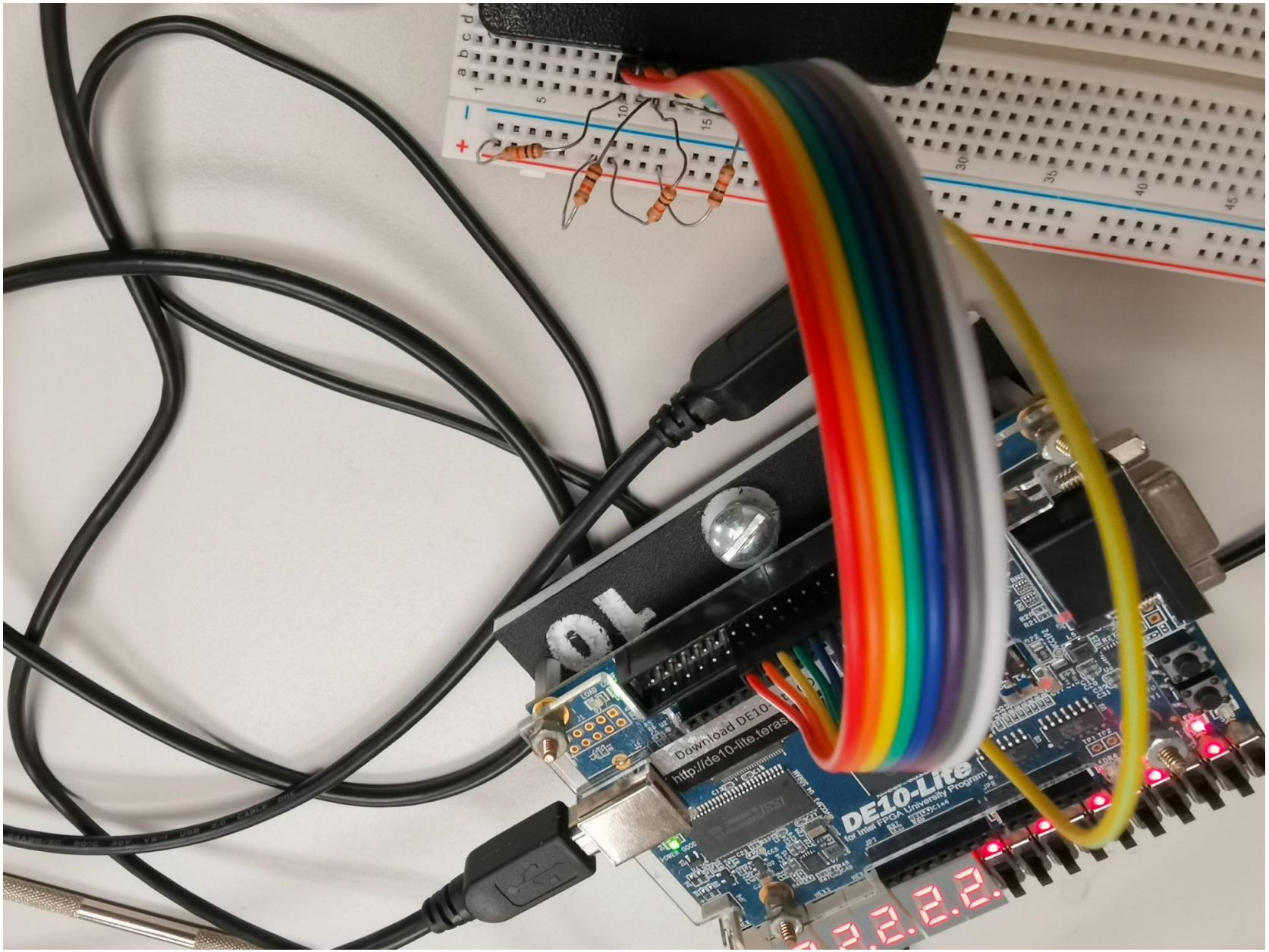


Results

We were able to print any text we want in the command shell,below is the picture that shows the result. As we can print whatever we want in the console



We were able to control the one led light corresponding to the switch below it. If we turn on the switch, the led stays on and if we turn it off, the led stays dark.


DE10-Lite
for Intel FPGA University Program

We did not get the keypad to print the desired number in the console with our code, we debugged the wire and tried to clear our logic but there was not enough time for us to reach the correct result for this part.

Conclusion

After we completed the experiments, we gained substantial insight into how microcontrollers can be used to process data. We learned about i2c communication, FPGAs, NIOS II and custom microcontrollers. In experiment 8 we figured out that by using polling instead of interrupts we can write from the PIC and Analog discovery to handle the logic of each byte received from the i2c bus. By writing a code with a poll function, we were finally able to create a logic analyzer that sends bytes of data. In experiment 9 we answered our other question from the introduction when we ignored the mismatched system ID and the mismatched system time. By ignoring the mismatched system ID and mismatched system time we were able to create a programming interface on an FPGA by running a Hello World sample script on it. For both experiments we were able to use logic analysis to compute data through the hardware interface of microcontrollers. We needed special software programs for the different experiments. We used Waveform to directly scope the outputs of experiment 8 with a PIC16F1769, while we used Quartas and Eclipse in experiment 9 to build a custom microcontroller with a NIOS 2 processor.

Appendix

Figure 8.1:

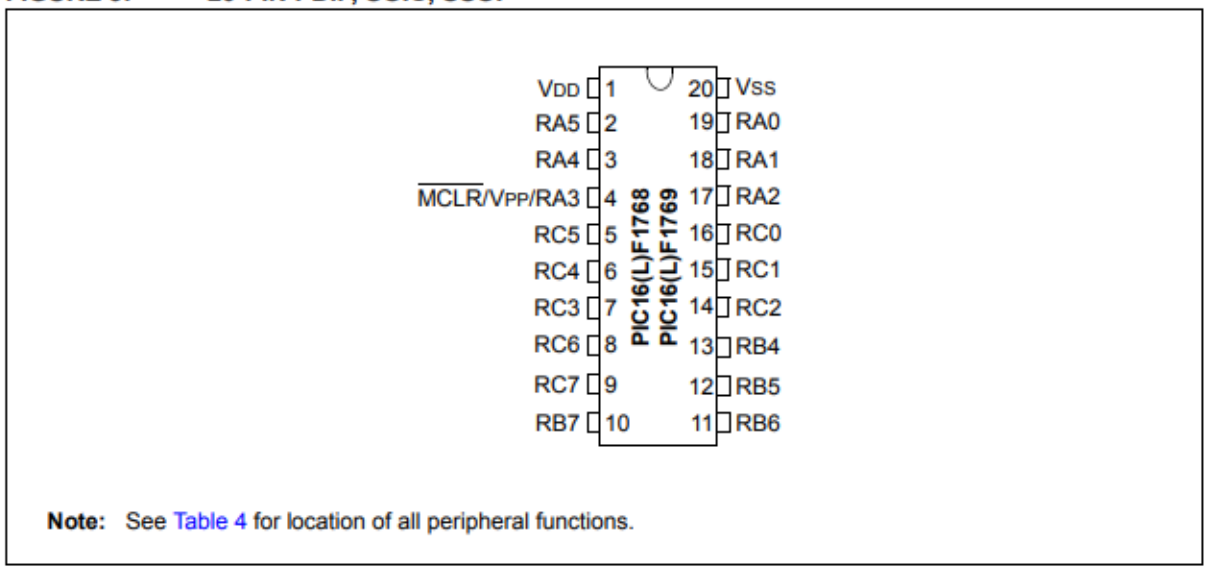
```
// CONFIG1

#pragma config FOSC = INTOSC    // Oscillator Selection Bits (INTOSC oscillator: I/O function on CLKIN pin)
#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = ON       // MCLR Pin Function Select (MCLR/VPP pin function is MCLR)
#pragma config CP = OFF         // Flash Program Memory Code Protection (Program memory code protection is disabled)
#pragma config BOREN = ON       // Brown-out Reset Enable (Brown-out Reset enabled)
#pragma config CLKOUTEN = OFF   // Clock Out Enable (CLKOUT function is disabled. I/O or oscillator function on the CLKOUT pin)
#pragma config IESO = ON        // Internal/External Switchover Mode (Internal/External Switchover Mode is enabled)
#pragma config FCMEN = ON       // Fail-Safe Clock Monitor Enable (Fail-Safe Clock Monitor is enabled)

// CONFIG2
#pragma config WRT = OFF        // Flash Memory Self-Write Protection (Write protection off)
#pragma config PPS1WAY = ON     // Peripheral Pin Select one-way control (The PPSLOCK bit cannot be cleared once it is set by software)
#pragma config ZCD = OFF        // Zero-cross detect disable (Zero-cross detect circuit is disabled at POR)
#pragma config PLLEN = ON       // Phase Lock Loop enable (4x PLL is always enabled)
#pragma config STVREN = ON      // Stack Overflow/Underflow Reset Enable (Stack Overflow or Underflow will cause a Reset)
#pragma config BORV = LO        // Brown-out Reset Voltage Selection (Brown-out Reset Voltage (Vbor), low trip point selected.)
#pragma config LPBOR = OFF      // Low-Power Brown Out Reset (Low-Power BOR is disabled)
#pragma config LVP = ON         // Low-Voltage Programming Enable (Low-voltage programming enabled)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>
/*
 *
 */

void i2c_main_init(void)
{
    TRISCbits.TRISC0 = 1; // Make i2c pin C0 input
    TRISCbits.TRISC1 = 1; // Make i2c pin C1 input
    ANSELCbits.ANSC0 = 0; // Make C0 digital not analog
    ANSELCbits.ANSC1 = 0; // Make C1 digital not analog
    SSP1CON1 = 0x28; // Enable and choose i2c main, 0x28 = 0b00101000
    SSP1ADD = 79; // clock frequency = Fosc/ (4*(SSP1ADD+1))
    SSP1STAT = 0x80; // Standard 100kHz speed, disable slew rate
    SSPCLKPPS = 0x10; // PPS input make SCL on C0
    SSPDATPPS = 0x11;
    // PPS input make SDA on C1
    RC0PPS = 0x12; // PPS output make SCL on C0
    RC1PPS = 0X13; // PPS output make SDA on C1
    PIR1bits.SSP1IF = 0; // clear I2C flag
}

void i2c_main_wait() {
    while(!PIR1bits.SSP1IF) {}
    PIR1bits.SSP1IF = 0;
}

void i2c_main_start() {
    SSP1CON2bits.SEN = 1;
    i2c_main_wait();
}

void i2c_main_stop() {
    SSP1CON2bits.PEN = 1;
    i2c_main_wait();
}

void i2c_main_write_data(int d) {
    SSP1BUF = d;
    i2c_main_wait();
}

void i2c_main_write(int a, int d)
{
```

```
    i2c_main_start();
    i2c_main_write_data(a); // Send the secondary address
    i2c_main_write_data(d); // Send data to the secondary
    i2c_main_stop();
}

/*
void i2c_secondary_init(void) {
    TRISCbits.TRISC0 = 1; // Make i2c pin C0 input
    TRISCbits.TRISC1 = 1; // Make i2c pin C1 input
    ANSELCbits.ANSC0 = 0; // Make C0 digital not analog
    ANSELCbits.ANSC1 = 0; // Make C1 digital not analog
    SSPCLKPPS = 0x10; // PPS output make SCL on C0
    SSPDATPPS = 0x11; // PPS output make SDA on C1
    RC0PPS = 0x12; // PPS output make SCL on C0
    RC1PPS = 0x13; // PPS output make SDA on C1
    SSP1STAT = 0x80; // Standard 100kHz speed, disable slew rate
    SSP1ADD = 0x92; // Secondary Address * 2 = 0x92 (Secondary Address = 0x49)
    SSP1CON1 = 0x36; // Enable, release clock, and choose i2c secondary with 7-bit address
    SSP1CON2 = 0x01; //Enable clock stretching
    PIR1bits.SSP1IF = 0; // clear I2C flag
}

void i2c_secondary_poll(int p1, int p2) {
    int i2c_state;
    int temp;
    int data;
    i2c_state = SSP1STAT & 0b00101101; // Zero all bits except D_nA, S, R_nW, and BF bits
    switch(i2c_state) {

        // Main sent the start bit and a write command. The buffer is full and has an address
        case 0b00001001: // D_nA=0 S=1 R_nW=0 BF=1
            temp = SSP1BUF;
            break;

        // Main sent the start bit and a write command. The buffer is full and has data
        case 0b00101001: // D_nA=1 S=1 R_nW=0 BF=1
            data = SSP1BUF;
            break;

        // Main sent the start bit and a read command. The buffer is full and has an address
        case 0b00001101: // D_nA=0 S=1 R_nW=1 BF=1
            temp = SSP1BUF;
            // Because it was a read command, you need to send data back to the Main
            while(SSP1STATbits.BF) {}
            SSP1BUF = p1;
            break;

        // When you send data back, the SSP1IF flag is set. No start, nothing in the buffer, still read and still data
        case 0b00100100: // D_nA=1 S=0 R_nW=1 BF=0
            break;

        // Main sent the start bit because it is requesting more than one byte of data, still data but R_nW undetermined
        case 0b00101000: // D_nA=1 S=1 R_nW=0 BF=1
        case 0b00101100: // D_nA=1 S=1 R_nW=1 BF=1
            // Send another byte
            if (!SSP1CON1bits.CKP) {
                SSP1BUF = p2;
            }
            break;

        // Any other state is due to an error, errors could be handled by checking error btis.
        default:
            while(1) {}
            break;
    }
    SSP1CON1bits.CKP = 1;
}
*/
void main(void)
{
    OSCCON = 0x70; // = 0b01110000
    i2c_main_init();

    int a = 0x5E; //94; // 47*2
    int d = 0x1B; //27; // 27*2
    i2c_main_write(a, d);

    while(1) {

        i2c_main_write(a, d);
```

```
        for(int i=0; i<10000; i++) {} //Delay so that we can see it better

        d++;

        /*
        // Every communication the Master starts,
        // sets the SSP1IF flag
        if (PIR1bits.SSP1IF) {
            PIR1bits.SSP1IF = 0;
            i2c_secondary_poll(245, 37); // i2c_secondary_poll(d1, d2)
        }
        */
    }

    return;
}
```

Figure 8.2

FIGURE 3: 20-PIN PDIP, SOIC, SSOP



Note: See Table 4 for location of all peripheral functions.

Figure 8.3:

```
Start, 5E [ 2F | WR ] NAK, F2 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F3 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F4 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F5 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F6 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F7 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F8 NAK, Stop
Start, 5E [ 2F | WR ] NAK, F9 NAK, Stop
Start, 5E [ 2F | WR ] NAK, FA NAK, Stop
Start, 5E [ 2F | WR ] NAK, FB NAK, Stop
Start, 5E [ 2F | WR ] NAK, FC NAK, Stop
Start, 5E [ 2F | WR ] NAK, FD NAK, Stop
Start, 5E [ 2F | WR ] NAK, FE NAK, Stop
Start, 5E [ 2F | WR ] NAK, FF NAK, Stop
Start, 5E [ 2F | WR ] NAK, 00 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 01 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 02 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 03 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 04 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 05 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 06 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 07 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 08 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 09 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 0A NAK, Stop
Start, 5E [ 2F | WR ] NAK, 0B NAK, Stop
Start, 5E [ 2F | WR ] NAK, 0C NAK, Stop
Start, 5E [ 2F | WR ] NAK, 0D NAK, Stop
Start, 5E [ 2F | WR ] NAK, 0E NAK, Stop
Start, 5E [ 2F | WR ] NAK, 0F NAK, Stop
Start, 5E [ 2F | WR ] NAK, 10 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 11 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 12 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 13 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 14 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 15 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 16 NAK, Stop
Start, 5E [ 2F | WR ] NAK, 17 NAK, Stop
```

Figure 8.4:

```
// CONFIG1

#pragma config FOSC = INTOSC    // Oscillator Selection Bits (INTOSC oscillator: I/O function on CLKIN pin)
#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = ON       // MCLR Pin Function Select (MCLR/VPP pin function is MCLR)
#pragma config CP = OFF         // Flash Program Memory Code Protection (Program memory code protection is disabled)
#pragma config BOREN = ON       // Brown-out Reset Enable (Brown-out Reset enabled)
#pragma config CLKOUTEN = OFF   // Clock Out Enable (CLKOUT function is disabled. I/O or oscillator function on the CLKOUT pin)
#pragma config IESO = ON        // Internal/External Switchover Mode (Internal/External Switchover Mode is enabled)
#pragma config FCMEN = ON       // Fail-Safe Clock Monitor Enable (Fail-Safe Clock Monitor is enabled)

// CONFIG2
#pragma config WRT = OFF        // Flash Memory Self-Write Protection (Write protection off)
#pragma config PPS1WAY = ON     // Peripheral Pin Select one-way control (The PPSLOCK bit cannot be cleared once it is set by software)
#pragma config ZCD = OFF        // Zero-cross detect disable (Zero-cross detect circuit is disabled at POR)
#pragma config PLLEN = ON       // Phase Lock Loop enable (4x PLL is always enabled)
#pragma config STVREN = ON      // Stack Overflow/Underflow Reset Enable (Stack Overflow or Underflow will cause a Reset)
#pragma config BORV = LO        // Brown-out Reset Voltage Selection (Brown-out Reset Voltage (Vbor), low trip point selected.)
#pragma config LPBOR = OFF      // Low-Power Brown Out Reset (Low-Power BOR is disabled)
#pragma config LVP = ON         // Low-Voltage Programming Enable (Low-voltage programming enabled)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>
```

```
/*
 *
 */

/*
void i2c_main_init(void)
{
    TRISCbits.TRISC0 = 1; // Make i2c pin C0 input
    TRISCbits.TRISC1 = 1; // Make i2c pin C1 input
    ANSELCbits.ANSC0 = 0; // Make C0 digital not analog
    ANSELCbits.ANSC1 = 0; // Make C1 digital not analog
    SSP1CON1 = 0x28; // Enable and choose i2c main, 0x28 = 0b00101000
    SSP1ADD = 79; // clock frequency = Fosc/ (4*(SSP1ADD+1))
    SSP1STAT = 0x80; // Standard 100kHz speed, disable slew rate
    SSPCLKPPS = 0x10; // PPS input make SCL on C0
    SSPDATPPS = 0x11;
    // PPS input make SDA on C1
    RC0PPS = 0x12; // PPS output make SCL on C0
    RC1PPS = 0X13; // PPS output make SDA on C1
    PIR1bits.SSP1IF = 0; // clear I2C flag
}

void i2c_main_wait() {
    while(!PIR1bits.SSP1IF) {}
    PIR1bits.SSP1IF = 0;
}

void i2c_main_start() {
    SSP1CON2bits.SEN = 1;
    i2c_main_wait();
}

void i2c_main_stop() {
    SSP1CON2bits.PEN = 1;
    i2c_main_wait();
}

void i2c_main_write_data(int d) {
    SSP1BUF = d;
    i2c_main_wait();
}

void i2c_main_write(int a, int d)
{
    i2c_main_start();
    i2c_main_write_data(a); // Send the secondary address
    i2c_main_write_data(d); // Send data to the secondary
    i2c_main_stop();
}
*/

void i2c_secondary_init(void) {
    TRISCbits.TRISC0 = 1; // Make i2c pin C0 input
    TRISCbits.TRISC1 = 1; // Make i2c pin C1 input
    ANSELCbits.ANSC0 = 0; // Make C0 digital not analog
    ANSELCbits.ANSC1 = 0; // Make C1 digital not analog
    SSPCLKPPS = 0x10; // PPS output make SCL on C0
    SSPDATPPS = 0x11; // PPS output make SDA on C1
    RC0PPS = 0x12; // PPS output make SCL on C0
    RC1PPS = 0x13; // PPS output make SDA on C1
    SSP1STAT = 0x80; // Standard 100kHz speed, disable slew rate
    SSP1ADD = 0x92; // Secondary Address * 2 = 0x92 (Secondary Address = 0x49)
    SSP1CON1 = 0x36; // Enable, release clock, and choose i2c secondary with 7-bit address
    SSP1CON2 = 0x01; //Enable clock stretching
    PIR1bits.SSP1IF = 0; // clear I2C flag
}

void i2c_secondary_poll(int p1, int p2) {
    int i2c_state;
    int temp;
    int data;
    i2c_state = SSP1STAT & 0b00101101; // Zero all bits except D_nA, S, R_nW, and BF bits
    switch(i2c_state) {

        // Main sent the start bit and a write command. The buffer is full and has an address
        case 0b00001001: // D_nA=0 S=1 R_nW=0 BF=1
            temp = SSP1BUF;
            break;

        // Main sent the start bit and a write command. The buffer is full and has data
        case 0b00101001: // D_nA=1 S=1 R_nW=0 BF=1
```

```
            data = SSP1BUF;
            break;

        // Main sent the start bit and a read command. The buffer is full and has an address
        case 0b00001101: // D_nA=0 S=1 R_nW=1 BF=1
            temp = SSP1BUF;
            // Because it was a read command, you need to send data back to the Main
            while(SSP1STATbits.BF) {}
            SSP1BUF = p1;
            break;

        // When you send data back, the SSP1IF flag is set. No start, nothing in the buffer, still read and still data
        case 0b00100100: // D_nA=1 S=0 R_nW=1 BF=0
            break;

        // Main sent the start bit because it is requesting more than one byte of data, still data but R_nW undetermined
        case 0b00101000: // D_nA=1 S=1 R_nW=0 BF=1
        case 0b00101100: // D_nA=1 S=1 R_nW=1 BF=1
            // Send another byte
            if (!SSP1CON1bits.CKP) {
                SSP1BUF = p2;
            }
            break;

        // Any other state is due to an error, errors could be handled by checking error bits.
        default:
            while(1) {}
            break;
    }
    SSP1CON1bits.CKP = 1;
}

void main(void)
{
    OSCCON = 0x70; // = 0b01110000

    /* Code for sig 1 & 2
    i2c_main_init();

    int a = 0x5E; //94; // 47*2
    int d = 0x1B; //27; // 27*2
    i2c_main_write(a, d);
    */

    i2c_secondary_init();

    while(1) {

        /* Code for sig 1 & 2
        i2c_main_write(a, d);

        for(int i=0; i<10000; i++) {} //Delay so that we can see it better

        d++;
        */

        ///* Code for sig 3 & 4
        // Every communication the Master starts,
        // sets the SSP1IF flag
        if (PIR1bits.SSP1IF) {
            PIR1bits.SSP1IF = 0;
            i2c_secondary_poll(255, 0); //i2c_secondary_poll(85, 170); // i2c_secondary_poll(d1, d2)
        }
        //*/
    }

    return;
}
```

Figure 8.5:

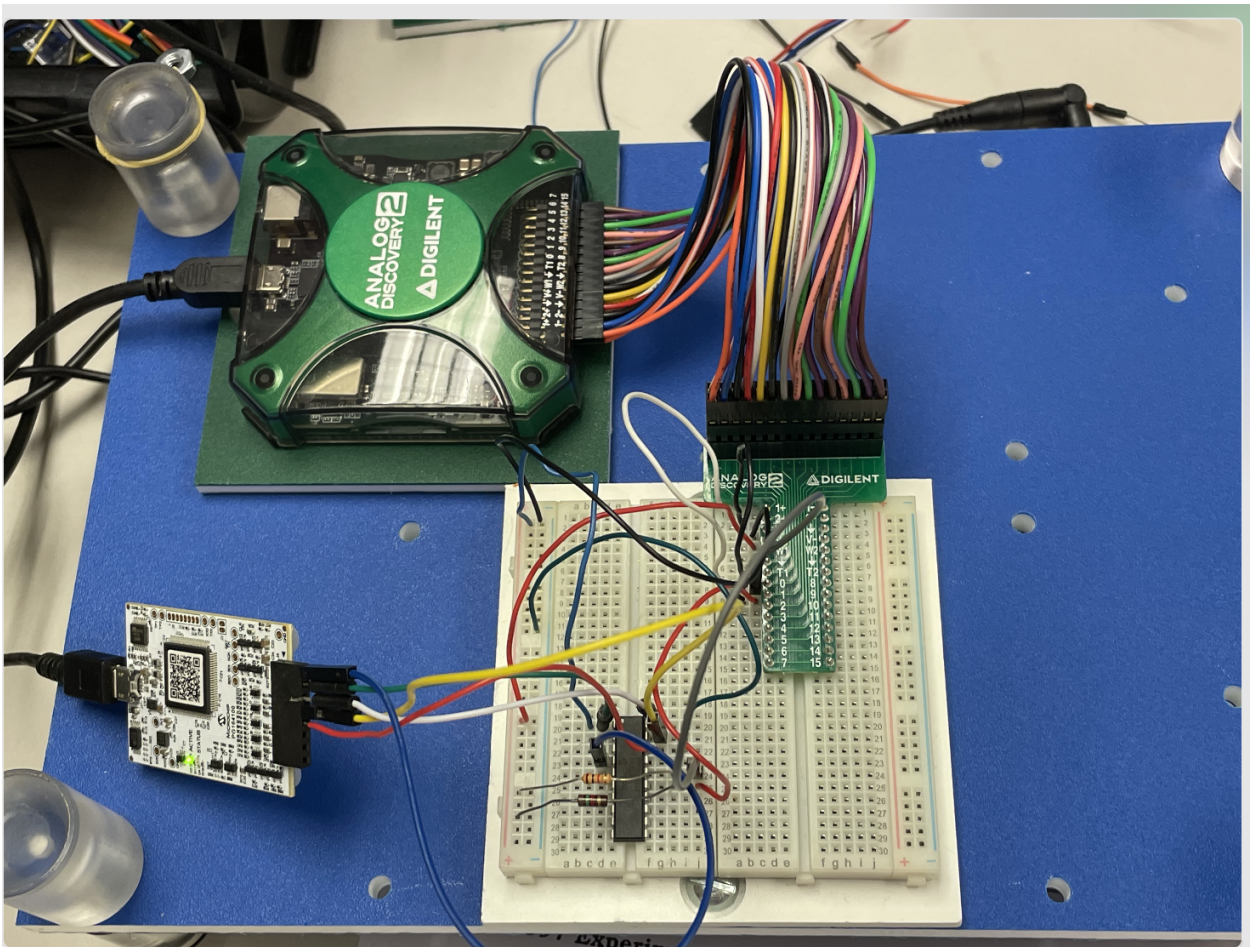


Figure 8.6:

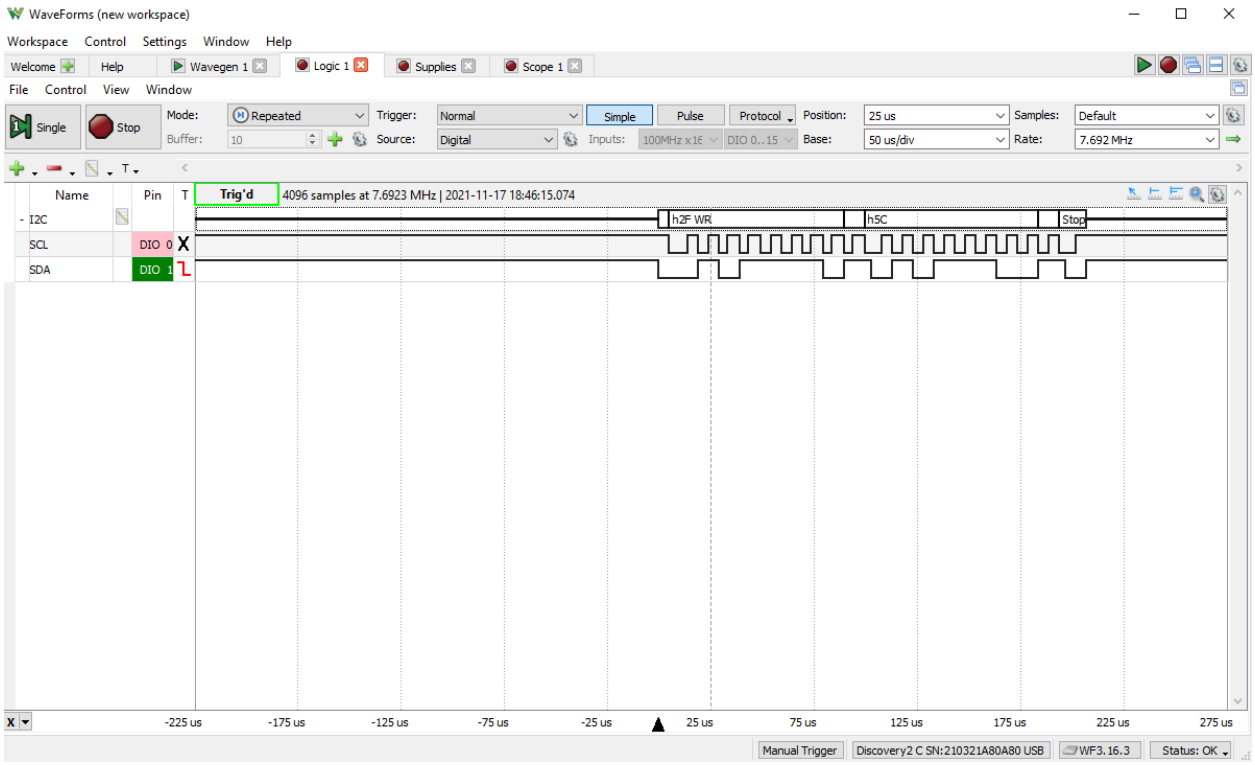

WaveForms (new workspace)
Workspace Control Settings Window Help
Welcome Help Wavegen 1 Logic 1 Supplies Scope 1
File Control View Window
Single Stop
Mode: Repeated
Buffer: 10
Trigger: Normal
Source: Digital
Simple Pulse Protocol
Inputs: 100MHz x16 DIO 0..15
Position: 25 us
Base: 50 us/div
Samples: Default
Rate: 7.692 MHz
Name Pin T
Trig'd
4096 samples at 7.6923 MHz | 2021-11-17 18:46:15.074
I2C
SCL DIO 0
SDA DIO 1
h2F WR
h5C
Stop
-225 us -175 us -125 us -75 us -25 us 25 us 75 us 125 us 175 us 225 us 275 us
Manual Trigger
Discovery2 C SN:210321A80A80 USB
WF3.16.3
Status: OK

Figure 8.7:

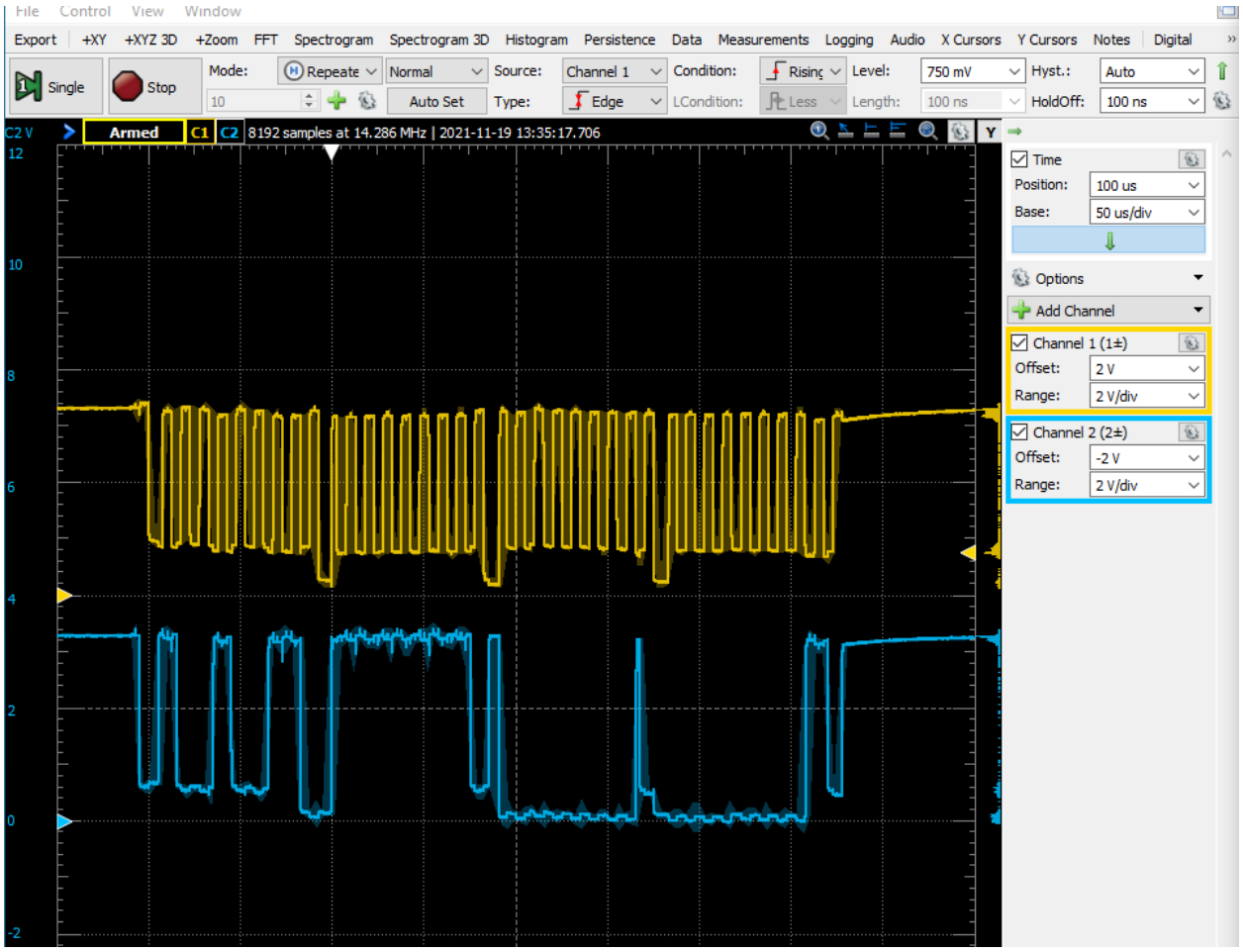
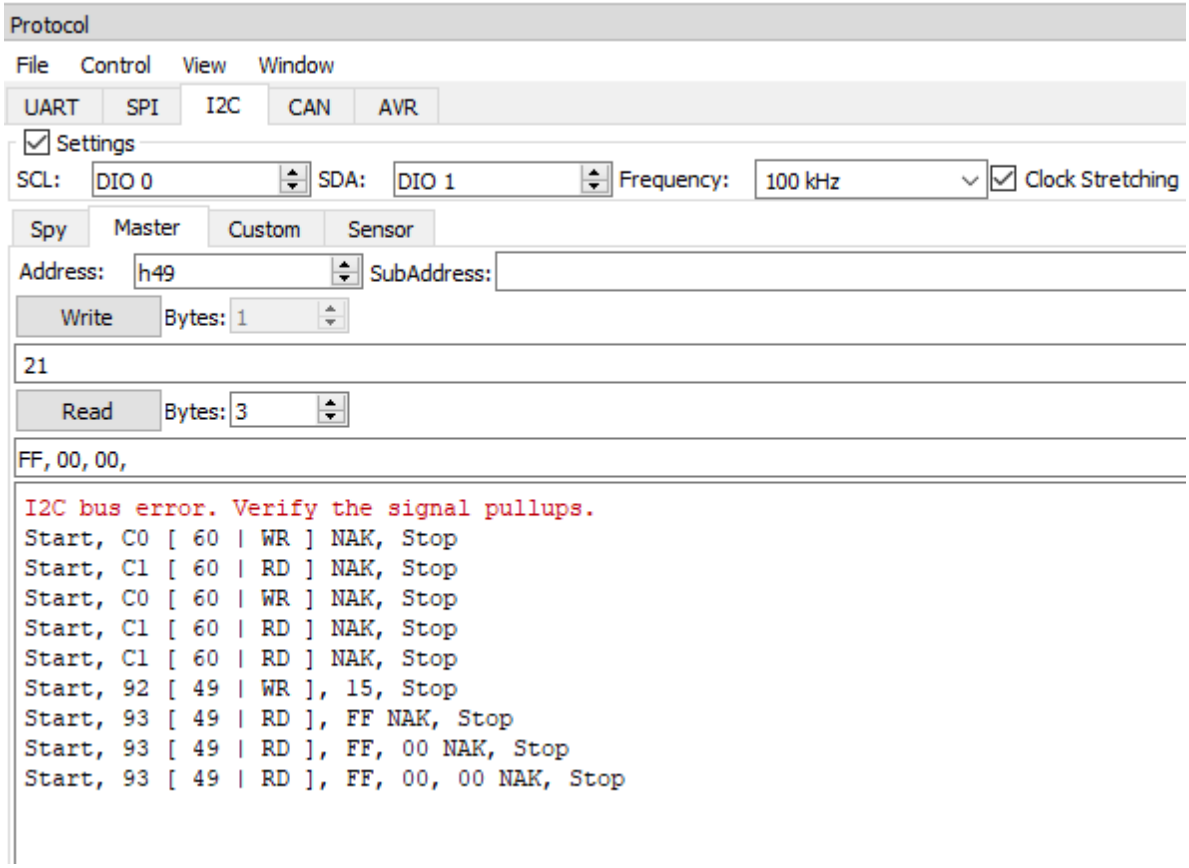

File Control View Window
Export +XY +XYZ 3D +Zoom FFT Spectrogram Spectrogram 3D Histogram Persistence Data Measurements Logging Audio X Cursors Y Cursors Notes Digital
Single
Stop
Mode:
Repeate
Normal
Source:
Channel 1
Condition:
Rising
Level:
750 mV
Hyst.:
Auto
10
Auto Set
Type:
Edge
LCondition:
Less
Length:
100 ns
HoldOff:
100 ns
C2 V
Armed
C1
C2
8192 samples at 14.286 MHz | 2021-11-19 13:35:17.706
Y
12
10
8
6
4
2
0
-2
Time
Position:
100 us
Base:
50 us/div
Options
Add Channel
Channel 1 (1±)
Offset:
2 V
Range:
2 V/div
Channel 2 (2±)
Offset:
-2 V
Range:
2 V/div

Figure 8.8:


Protocol
File
Control
View
Window
UART
SPI
I2C
CAN
AVR
Settings
SCL:
DIO 0
SDA:
DIO 1
Frequency:
100 kHz
Clock Stretching
Spy
Master
Custom
Sensor
Address:
h49
SubAddress:
Write
Bytes:
1
21
Read
Bytes:
3
FF, 00, 00,
I2C bus error. Verify the signal pullups.
Start, C0 [60 | WR] NAK, Stop
Start, C1 [60 | RD] NAK, Stop
Start, C0 [60 | WR] NAK, Stop
Start, C1 [60 | RD] NAK, Stop
Start, C1 [60 | RD] NAK, Stop
Start, 92 [49 | WR], 15, Stop
Start, 93 [49 | RD], FF NAK, Stop
Start, 93 [49 | RD], FF, 00 NAK, Stop
Start, 93 [49 | RD], FF, 00, 00 NAK, Stop