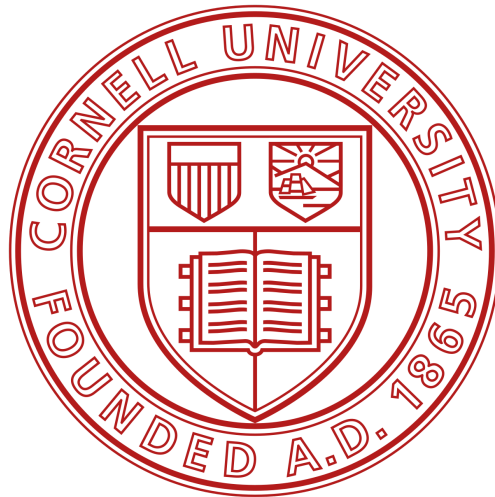


PIC32 AND RASPBERRY PI INTERFACE

A Design Project Report Presented to the School of Electrical and Computer Engineering of Cornell University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering



Submitted by Zesun Yang (zy366),
Advisors: Dr. Bruce Land, Dr. Joseph Skovira
Degree Date: Dec 2019

* Some content may be the same as the 2018-2019 PIC & Pi Interface report because the authors (Zesun Yang, Vipin Vengupal and Ye Kuang) wrote these together in the 2018 fall semester project report. This report is a continued work of the 2018-2019 project.

Abstract

Master of Engineering Program School of Electrical and Computer Engineering

Cornell University Design Project Report

**Authors: Zesun Yang (zy366)
with previous contributions of Vipin Venugopal (vv258), Ye Kuang (yk749)**

Abstract: This project is a continued work of the PIC32 and raspberry Pi Interface Project in the 2018-2019 Academic year in which I implemented the PIC and PI interface with Vipin and Ye. The goal of the project was to develop a cross-platform system capable of performing real-time tasks while enabling the development of software decoupled from the hardware and to implement a final application to demonstrate the system. The project aimed to combine the best of both worlds of Embedded Operating System in Raspberry Pi and hardware-dependent bare-metal firmware of PIC32 to build a low-cost oscilloscope. The project proved that the interface we developed last year was functionally sufficient to provide easy interfacing. I also addressed and fixed some issues I have encountered in the project development phase.

Executive Summary

The objective of this project was to develop a hybrid system that can provide optimum real-time performance at minimum development cost and to develop an end application to demonstrate the PIC and Pi combined system capabilities. A hybrid approach was adopted for building a system capable of carrying out hard real-time tasks using hardware parallelism available in a PIC32 microcontroller, while simultaneously using the higher level of abstraction provided by the embedded operating system in a Raspberry Pi, relieving the developer from taking care of the background tasks.

The tasks involved in the interfacing were to build a library and communication protocol between the PIC32 and Raspberry Pi, and developed an application using this interface. The designed and initial implementation of the library and communication protocol were made in the fall 2018 semester by me, Vipin Venugopal and Ye Kuang. Then the final implementation was done by Vipin Venugopal and Ye Kuang in the spring 2019 semester. The ultimate goal of this project was to utilize the finished communication protocols and libraries to make an end application. I chose to implement an oscilloscope that can display waveform as well as able to perform basic operations such as changing the time and voltage per division. The purpose of this design project is to prove that our previous work was functional and to fix any issues along the way of development to refine our system.

The result of this project was very successful. The oscilloscope was able to use ADC on the PIC32 to sample data and transmit to the Pi, and the Pi received the data and plotted it on the piTFT just if it's a scope display. It has a high measuring accuracy as well as fast sampling speed. In addition to the scope application, I was able to prove that our system is easy to use. I was able to obtain relevant information I needed for the oscilloscope application from the software and hardware guide provided by Vipin and Ye. In addition, found some issues with our libraries and corrected them.

Work Distributions

Item num	Task	Vipin Venugopal	Ye Kuang	Zesun Yang
2018 Fall Semester				
1	PIC32 peripheral selection and pin assignment	x		x
2	Mode of Communication and protocol design		x	x
3	State machine design for Communication	x	x	x
4	Firmware implementation on PIC for peripheral control	x		x
5	Python Library development for Raspberry Pi		x	
6	Communication Testing	x	x	x
2019 Spring Semester				
7	Complete and final firmware implementation on PIC for peripheral control	x		
8	Complete and final Python Library development for Raspberry Pi		x	
9	Module Testing of PIC32 firmware	x		
10	PCB design for PIC32 board	x		
11	Assembly and Testing of PIC32 board	x	x	

12	Application development for Raspberry Pi		x	
13	Module Testing for Raspberry Pi Application		x	
14	Integrated testing for PIC and Pi Interface with custom board	x	x	
2019 Fall Semester				
15	Reassembly and testing of PIC32 board			x
16	Oscilloscope application development			x
17	Module Testing for Raspberry Pi Application			x
18	Addressed and fix protocol & library issues			x

1 Introduction	6
2 Background	6
3. Design and Testing	9
3.1 PIC & Pi Oscilloscope Hardware Design and Testing	9
3.2 PIC & Pi Oscilloscope Software Design	10
4. Result	13
5. Issues	18
6. Future Work	20
6.1 Architecture redesign	20
6.2 More applications covering more modules	20
6.3 A better scope	21
7. Conclusion	21
8. Acknowledgment	22
9. Reference	22
Appendix A. Modified PIC32 Scope Code	23
Appendix B. Modified Raspberry Pi Interface Code	46
Appendix C. Oscilloscope Code	51

1 Introduction

The most common approach to build an embedded system is to use a microcontroller and write the code from the ground up. This approach is appropriate for the classic definition of an embedded system as a computer system capable of handling a specific function. But over the years, the applicability of the definition is diminishing as the embedded systems move closer to the general-purpose computers. Recently, there has been a migration towards OS-based microcontrollers, to reduce development time and effort and for ease of performing high-level tasks. But this comes at the cost of poor real-time performance. Developers need to consider the trade-offs between the development effort, time and performance in a real-time environment while choosing from these two approaches.

The objective of this project is to develop a hybrid system that can provide optimum real-time performance at minimum development cost. The goal is to develop an end application, which in this report, an oscilloscope to demonstrate these features.

2 Background

As the world progresses towards Internet of Things, the expectations from an embedded system are increasing exponentially. The devices are expected to provide connectivity, form part of sensor networks and respond to events and commands from other systems. Traditional bare-metal microcontrollers are lacking in this regard as building the software from scratch becomes increasingly difficult, as the trend progresses. The use of an operating system could provide a much higher level of abstraction, and relieve the developer of most of the background tasks from carrying out these tasks. Linux based operating systems come with a whole suite of free tools and libraries for supporting this. This is the major reason why a Linux OS based

microcontroller like Raspberry Pi has become a more popular choice for IoT applications when compared to bare-metal microcontrollers like PIC. However, the embedded systems used in safety-critical environments are expected to perform hard-real-time tasks. Hardware access at a level low enough to achieve this is often not available while using a traditional Operating System. While an OS-based microcontroller like Raspberry Pi does not do well with analog interfaces and real-time tasks, a bare-metal microcontroller like PIC32 does not offer the versatility of the OS and community support of Linux.

The idea is to use a Raspberry Pi which runs a Linux distribution to perform high-level tasks and provide connectivity and user interface, and then use the PIC32 microcontroller to perform low level and time-critical tasks. This requires a hardware-dependent firmware on the PIC32 capable of responding to commands, and application-specific software running on Raspberry Pi over the Linux OS and a high-speed interface between the two to enable the Pi to control the PIC. The command line between the Pi and PIC will be encapsulated into a library running on the Pi, which will abstract the functions of the PIC microcontroller and act as a driver for the embedded hardware. The application can perform low-level tasks by making library calls without worrying about the actual implementation.

The PIC32 microcontroller is a powerful, 32-bit CPU with the following peripherals

- 1) Analog to Digital Converter
- 2) Data Memory Access
- 3) Communication Interfaces
- 4) Timers
- 5) Output Compare Unit
- 6) Input Capture Unit
- 7) General Purpose I/O

Peripheral libraries are also available to control them. The downside of the PIC32 microcontroller is that it has no operating system and code has to be written from start, also rendering it hardware dependent.

The Raspberry Pi runs a full Linux distribution and it comes with many features like

- 1) Bluetooth
- 2) LAN
- 3) Wi-Fi connectivity
- 4) USB and serial channel
- 5) HDMI camera and display-port interface

However, the Pi does not perform real-time tasks very well. The combination will have the best of both worlds. Both the devices are combined to take advantage of the PIC32 peripherals for interfacing, and the high performance of Pi on computation. The plan is to use the PIC32 for input reading, output generation, and use the Pi for decision making and user-interface. The resultant system would have a master-slave configuration with Raspberry Pi as Master and PIC32 as the slave, and command response interface between the two.

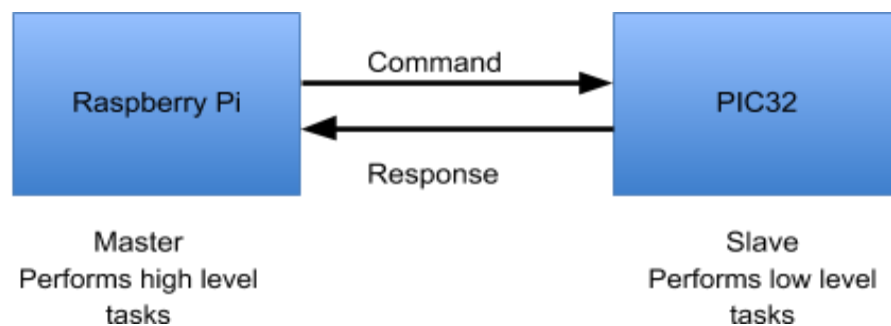


Figure 1: Proposed System

The application I chose to implement is an oscilloscope. The PIC32 has 5 ADC pins which is sufficient enough to do analog data sampling. The raspberry Pi has plenty of memory spaces and it has Python platform which provides abundant helpful libraries that allows high-level application development.

3. Design and Testing

The following contents are not included in the design section:

- a. PIC32 peripheral selection and pin assignment
- b. Mode of Communication and Protocol design
- c. State machine design for Communication
- d. Firmware implementation on PIC for peripheral control
- e. Python Library for Raspberry Pi

Because they are already covered in the 2018-2019 PIC and Pi Interface report. See

https://people.ece.cornell.edu/land/courses/eceprojectsland/STUDENTPROJ/2018to2019/vv258_yk749/project_139_report.pdf for details.

However, the design and testing section does cover:

- a. Scope hardware design
- b. Scope software design

3.1 PIC & Pi Oscilloscope Hardware Design and Testing

The hardware of this oscilloscope project design consisted of a PIC32, a raspberry Pi 3, a PiTFT display, a button, and a potentiometer. The PIC32 was used to acquire voltage data, the raspberry Pi 3 was used to plot the data and display on the PiTFT. The 4 buttons on the piTFT display were used for changing voltage per division, time per division. The external button was used as a bail-out button wired to GPIO4 to exit from the application without using keyboard interrupts and clear the GPIOs. The potentiometer was wired to a 3.3V pinout from the PIC32. It was used for testing and debugging the oscilloscope on DC inputs. The circuit schematic is shown in Figure 1. The hardware was tested by first verifying the voltage pins on the PIC board were able to produce expected voltage, then the programmer was able to program the PIC32. The next step

was verifying the connection between the PIC32 and the Raspberry Pi. I connected the debug serial port and saw there were data flow between these two. The bail-out button was tested by successfully exiting from an infinitely-running user program and saw no errors such as ‘GPIOs were busy or in use’ showed up after the bail-out. The complete schematic is shown in Figure 2.

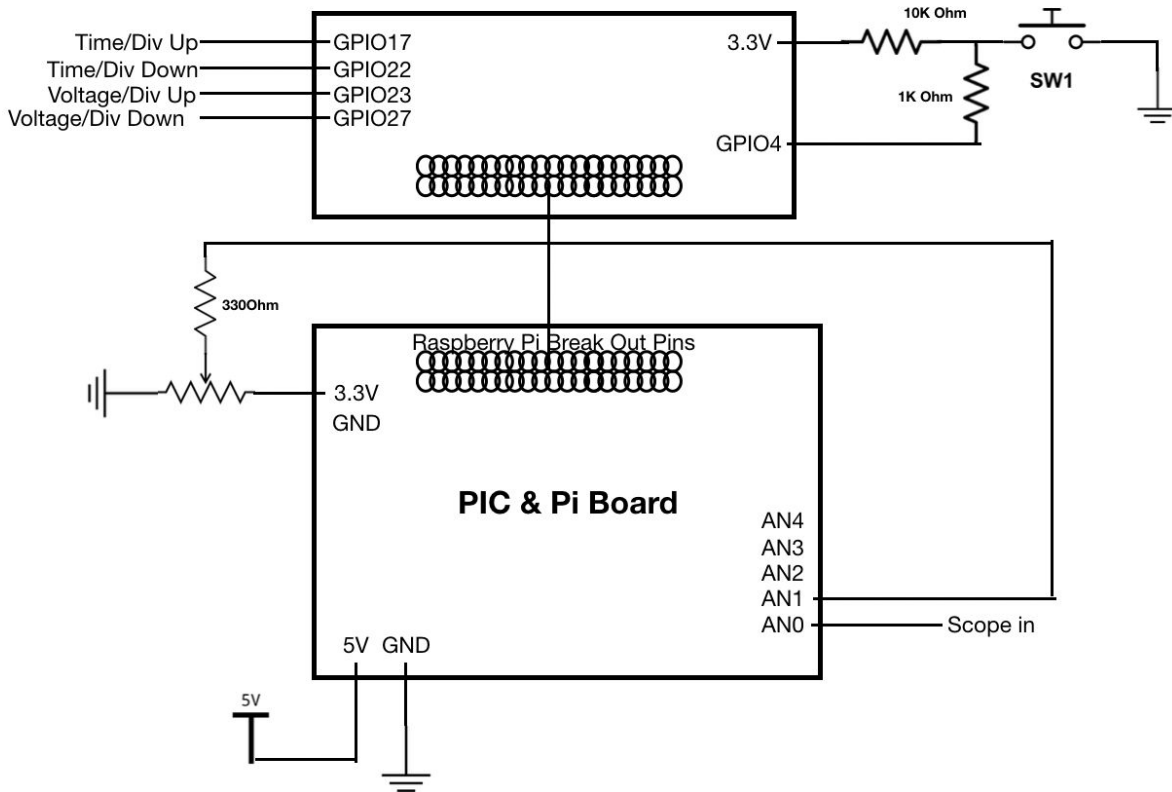


Figure 2. Oscilloscope Schematics

3.2 PIC & Pi Oscilloscope Software Design

The design of the software can be divided into 3 parts, PIC32 software design, PIC and Pi communication, and Pi software design.

The PIC32 runs at 40MHZ CPU clock rate. The most important component in the oscilloscope is the ADC which is responsible for data acquiring and data sampling. Initially, I have enabled all 5 ADCs on the PIC32. The ADC is configured to perform auto triggering and auto sampling which ensure it's sampling as fast as possible if the sample rate is not specified by the user. The offset test is disabled, the scan mode is enabled and it processes 4 samples at a time using a dual buffer and multiplexer A. Whenever we set the sampling frequency, we open timer 1. The ADC reading ISR uses timer 1 and reads the ADC values into a set buffer. Later on, I changed the ADC configurations and only enabled 1ADC because in an oscilloscope application, I will only be using 1 scope probe. By disabling unused ADC channels, it saves time by not scanning the data on idle channels. In addition, I set the ADC to only use single buffer and do 1 sample at a time.

The PIC32 and raspberry Pi communication were accomplished through UART. Although it seemed a poor choice because UART is slow, the SPI was already in use for the piTFT display, therefore UART is the next best option. In addition, when our team first designed the communication protocol, we were settled for UART communication because SPI requires more pins versus UART only requires 2 wires. I2C requires 2 wires as well, but the implementation of i2C can be complex. SPI communication can only be initiated by the master device while UART can be bidirectional, which is exactly what we wanted because we would like to have the PIC32 and raspberry Pi talk to one another. The communication rate was set to be 19200 on both the PIC32 and the Raspberry Pi. The communication was verified by connecting the UART debug serial to the PIC and the Pi separately, and see data flow on both of them including the title "Welcome to the PIC and Pi Interface".

The Raspberry Pi was mainly used to send commands to the PIC32 and parse data transmitted by the PIC. The Raspberry sent commands to the PIC to set the sampling frequency based on time/div information, and then specified which ADC channel and buffer to use. The ADC data sent back range from 0-1024 because ADC is 10 bits, and I rescaled it to match our 0-3.3V voltage level. For testing, I first wired the potentiometer output values to ADC1 and plotted the result. An example image is shown in Figure 3.

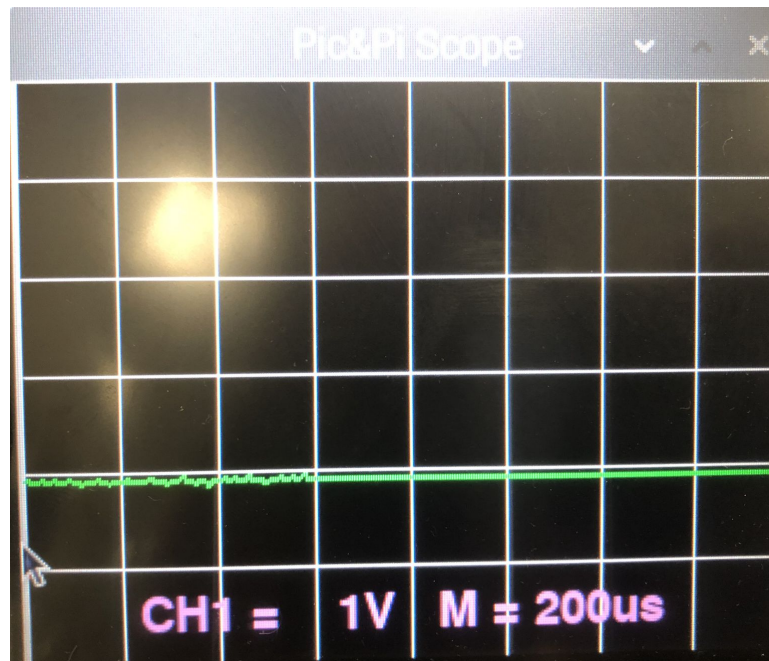


Figure 3. DC input plotted by the scope

The oscilloscope supports 6 time/div settings, they are '200us','500us',' 1ms',' 2ms',' 10ms','20ms' respectively. It also supports 4 different voltage/div settings, they are ' 2V',' 1V','500mV','250mV' respectively. The Raspberry used pygame library to draw the scope grid and plot the waveform. I had a predefined array storing the grid's x-position and the y positions. The grid was drawn using the pygame.draw.line function. I have also used a flag called 'done' to indicate whether the user has clicked the close button on the oscilloscope GUI. The program can closed on either the close button is clicked, or the bail-out button is pressed for about a second. Because the (0,0) point is at the top left-hand corner, when I plotted the data I need to 'inverse' the coordinates by performing subtraction from the scope display height. This was verified by comparing the PIC & Pi oscilloscope waveform and the Tektronix waveform of the same source. A red dot at the left side of the grid indicates the 0-level of the scope.

4. Result

The interface we designed last year worked successfully. It can provide easy interfacing between the PIC32 and the raspberry Pi 3, and leave enough pins for different purposes for the users to explore, combine, and develop interesting applications. There are some minor issues with our previous work, but these issues were not detrimental. These issues will be discussed in section 5. The hardware is shown in Figure 4 and Figure 5.



Figure 4. Side view of the raspberry Pi, PIC board and the piTFT stack

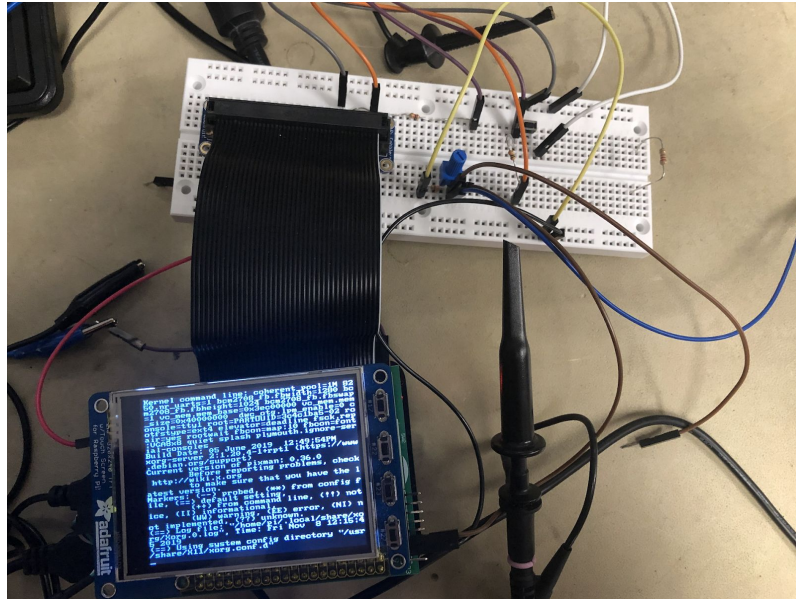


Figure 5. Top view of the circuit connection

As I mentioned in section 3, the scope supports 6 different time-per-division settings. Figure 6-11 shows the results of the same saw wave with different time settings.

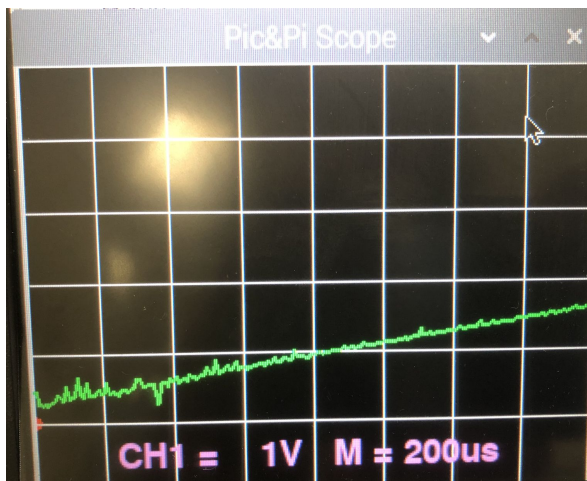


Figure 6. Saw with 1V, 50us setting

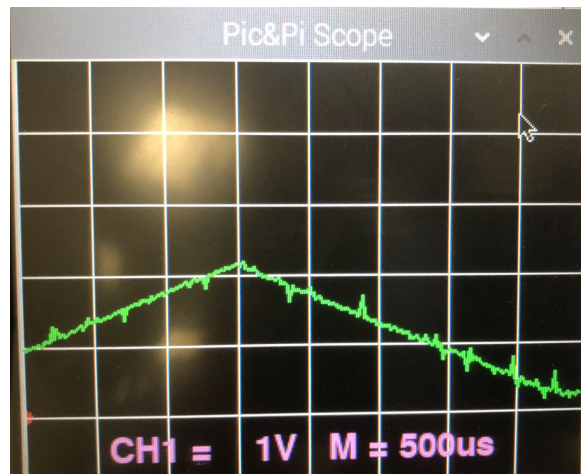


Figure 7. Saw with 1V, 500us setting

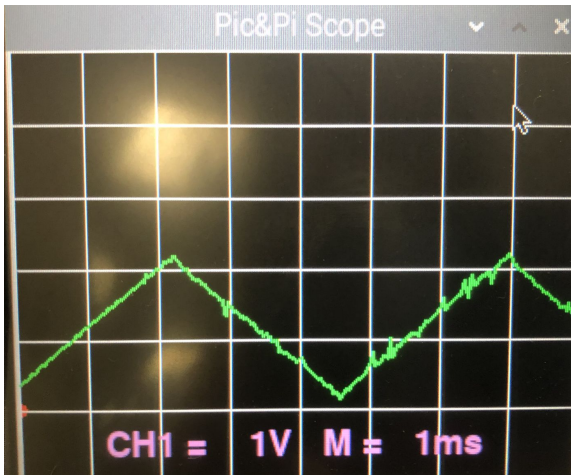


Figure 8. Saw with 1V, 1ms setting

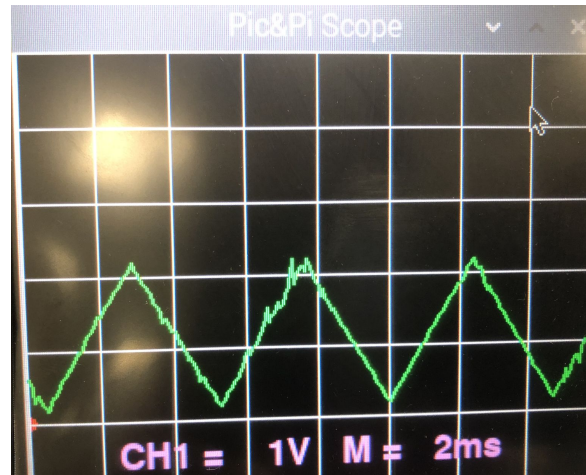


Figure 9. Saw with 1V, 2ms setting

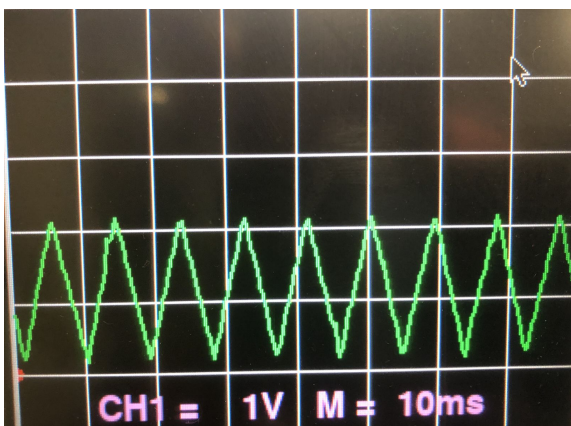


Figure 10. Saw with 1V, 10ms setting

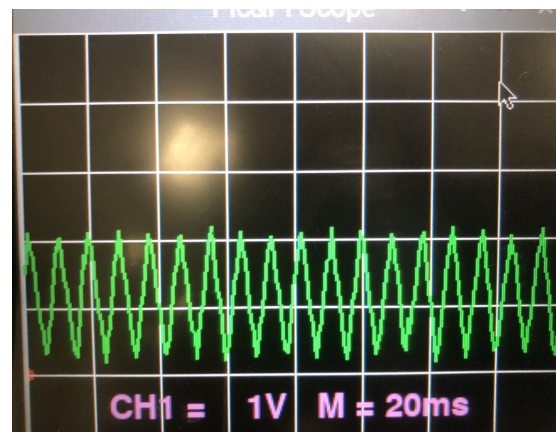


Figure 11. Saw with 1V, 20ms setting

The scope also supports 4 different voltage display settings. Figure 12-14 shows the same saw waveform with different voltage display settings. (voltage = 1 is shown in Figure 10 already). Figure 15 shows the Tektronix scope display of the same saw waveform. The oscilloscope I developed with the interface we built last year worked very well. As one may see in Figure 15, the PIC and PI scope has the same value-display as an a-thousand-dollar scope. Table 1 on page 17 also proves the scope has low measuring error compared to the real input.

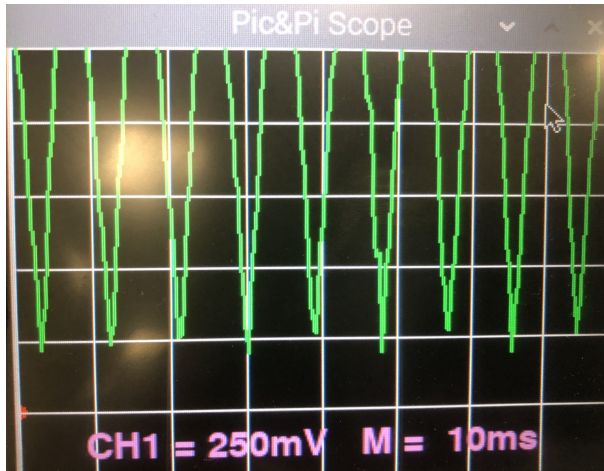


Figure 12. Saw with 0.25V, 10 ms setting

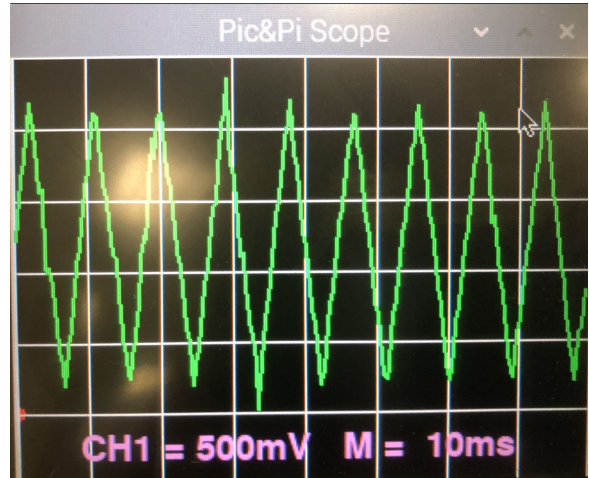


Figure 13. Saw with 0.5V, 10 ms setting

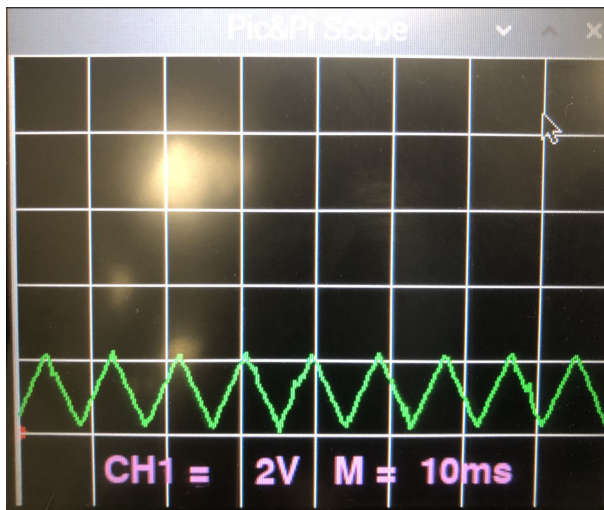


Figure 14. Saw with 2V, 10ms setting

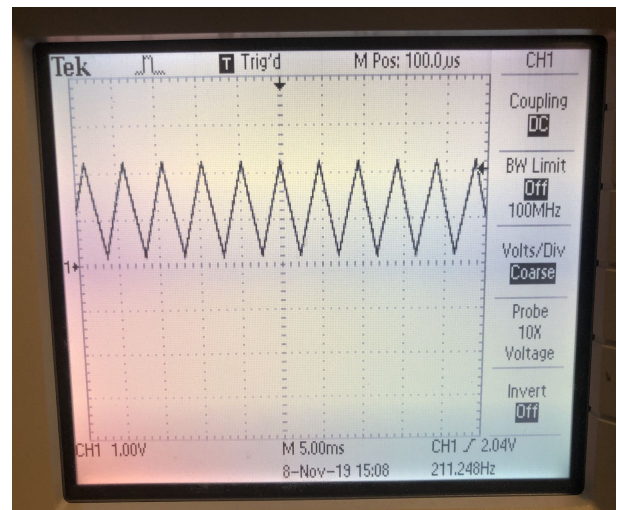


Figure 15. Tektronix scope display

The scope can capture all sorts of waveforms. Figure 16 shows a sine wave, figure 17 shows the Tektronix capture of the same sinewave; figure 18 shows a square wave, and figure 19 shows the Tektronix capture of the same square wave. To test the accuracy of the scope, I took a measurement of a 0.88V DC input. The table below shows the result of 4 consecutive readings of the scope (the readings are all about the same values, the 4 consecutive samples were arbitrarily chosen in all readings). The scope has a good accuracy that the measuring error is lower than 0.5%.

Table 1. 4 Consecutive Measurements on 0.88 V DC

	Scope reading	Percentage error
Measurement 1	0.8766	0.38%
Measurement 2	0.8765	0.39%
Measurement 3	0.8829	0.32%
Measurement 4	0.8830	0.34%

The PIC and Pi scope also has a short reaction time. The reaction time is based on how fast the PIC acquires the data, how fast the data transmission and acknowledge happens and how fast the PI plots the data. I measured the reaction time by printing out the elapsed time between the data sampling and actually seeing the plot, and the reaction time i got was within 0.171-0.182 second. This short reaction time implies our protocol was good and efficient.

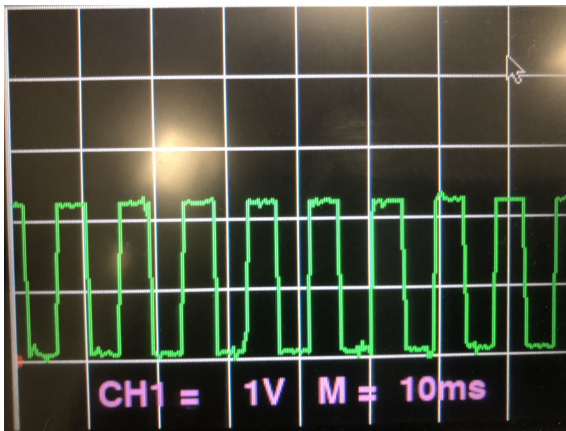


Figure 16. PIC & Pi scope - sine wave

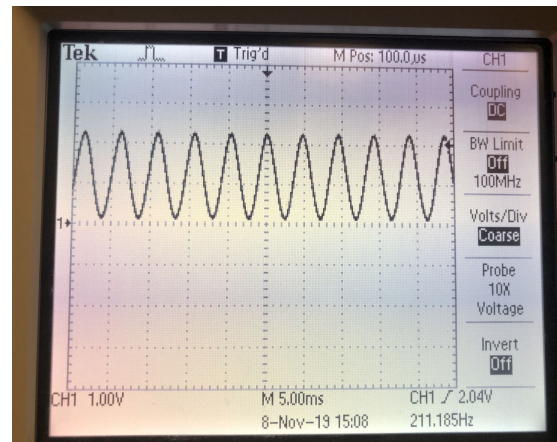


Figure 17. Tektronik scope - sine wave

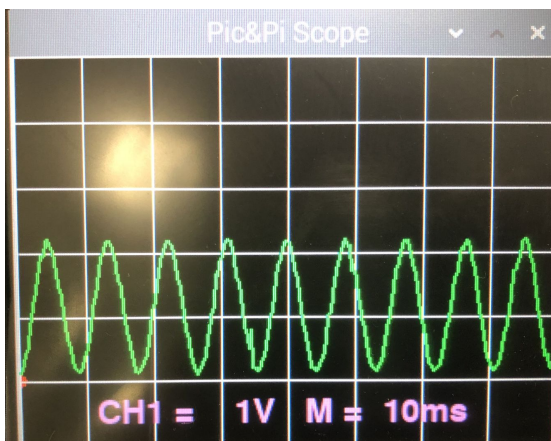


Figure 18. PIC & Pi scope - square wave

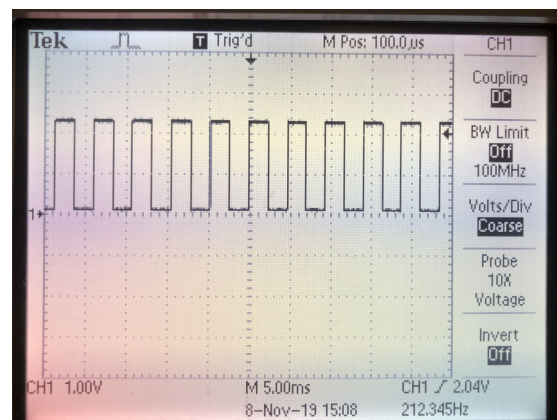


Figure 19. Tektronik scope - square wave

5. Issues

I have encountered some issues with developing the oscilloscope application, and these were due to the initial design of the protocol architecture and implementation. However, these issues are not detrimental. They are hard to find depending on what modules the application is using. Since the application developed last year did not use the ADC, I assumed limited testing has been done for the ADC modules. It is beneficial to have a separate application developed and catch these bugs. In this section, I will summarize 3 issues I have found.

The first issue I found was the use of unsigned char versus signed char for the buffers. In the PIC32 code of last year, we set all the buffers to be of type 'signed char'. The raspberry Pi initiates the data transfer by sending an openADC command as well as setting the sampling frequency, however, because the fact that the receiving buffer at the PIC side is signed, and we did not specify the data conversion between the PIC and the PI, the PIC pad the data and set a negative frequency which results in incorrect scope display. The Raspberry Pi also receives the acknowledge after the PIC sets the frequency, and display "setting -96K frequency" which is very confusing. This issue was resolved by having a separate unsigned char buffer for the sampling frequency. It ensured when we are sampling frequency, it will never get confused and set a negative value.

The second issue I found was the incomplete implementation of the check_buf function for both the PIC code and the raspberry Pi interface code. The symptom of having incomplete implementations of check_buf is the raspberry stopped asking for more data after a short time and it continues to execute the next line. At the beginning of data parsing, more than half the values of the Pi's buffer were 0s. After the first transfer, the buffer was filled with obsolete values/lagged values from the previous communication, and thus caused the huge inconsistency between real voltage values and values received by the Pi. The issue was resolved by complete the implementation of the check_buf function. On the PIC side, after it verifies the buffer status,

it should send the status back to the Pi immediately. On the Pi side, it should check the status values sent back from the PIC, and if the value does not match the case that every buffer is ready to send (hex 0x0F), it will stay in the loop and ask for more data instead of moving on.

The last issue I found was our interface architecture was designed to send only a byte at a time which was ineffective and not sufficient for some applications. In addition, the documentation on the readSample function is misleading. In Vipin and Ye's implementation, the max value one could read and parse in the transmission array was 255 (because it's the max value one byte can represent). If one wants to read a larger value than 255, he or she has to use the read2Byte function, and the value then gets split into 2 bytes like what Table 2 shows.

Table 2. Read1byte function and read2Byte function

Value	Transmission array of read 1 byte function	Transmission array of read 2 bytes function
199	[0xC7]	[0x00, 0xC7]
302	[0x255]	[0x36, 0xFF]

It becomes an issue when one tries to use the read2Byte function and the readBuffer function together. According to the documentation, the readBuffer function takes x number of samples one wants to read as an argument, and asks the slave device to send x number of samples. However that's not true because the readBuffer function actually asks slave device to send x number of bytes, not samples. Given that's what the function really does, if one wants to read 500 samples with values greater than 255, the person uses the read2Byte function and get in fact 250 valid samples back, not 500, while that master device thinks it has 500 samples and accesses all 500. The symptom of this would be the first 250 samples were correct and valid, and the rest 250 samples were junk data. I worked around the issue by only reading half of the samples even though the system was able to acquire all samples.

Despite the issue with the number of samples, application wise I think it was not efficient enough to send only 1 byte at a time. Take the scope application for an example, the ADC can read values ranging from 0 to 1024, but one byte can only represent value 0-255. Luckily we already had the read2Byte function which covers 0-65535 which is sufficient for the scope application. However, what if a user wants to read a value greater than 65535? Writing a read3Byte function does not seem to make sense. The readxByte functions are not adaptive, and I see that as an issue of the interface.

Lastly, one minor flaw I found is that we cannot use the debug feature when performing fast, real-time operations. Printing debug messages to the serial terminal significantly slow down the data acquisition as well as data transfer. Take the PIC and PI oscilloscope for an example, enabling the serial debug will result in a distorted and incorrect waveform.

6. Future Work

Vipin Venugopal and Ye Kuang mentioned some future work in their report. So here I will add some more things that can be improved instead of discussing things they already covered.

6.1 Architecture redesign

As discussed in section 5, the architecture can be better. The most important changes that I believe should be done is to have our interface take more bytes at a time without wasting samples. In the case of oscilloscope development, I only need to process up to 1024 at a time. Even the read2Byte function seemed to work for me, but it wastes half of the samples. If we want to support wider range applications, we should redesign our communication protocol so we are not tightly limited by the values we are transmitting, and we are not wasting samples.

6.2 More applications covering more modules

I found issues with the sign of buffers, incompleteness of functions because I was developing a different application using different modules compared to what was done last year. The issues and bugs are hidden until one tries to use these modules. Therefore I think it will be great that I can develop different applications using different modules, or even combine different modules and see how well they perform together to discover potential bugs and refine our interface.

6.3 A better scope

The scope I made has only limited functionalities. For future work, I can add a fake trigger to imitate the trigger on the Tektronix scope. I can also add a few lines of code to be able to move the 0-level on the scope and thus move the waveform up and down. If I have more time, I can also make a function generator using the PIC and Pi interface. There are many things that can make the scope cooler.

7. Conclusion

This project is very successful in two aspects. The first and most important aspect being the success of the PIC and Pi interface. It is easy to use and provides the possibilities to utilize both the PIC32's real-time operating speed, its many peripherals as well as the Raspberry Pi's high-level abstraction and storage space. The interface will be very useful for embedded developers. It has the best of two worlds for a cheap cost. The second aspect is the success of oscilloscope development. It has great measuring accuracy and it has a short reaction time. It's portable and cheap for electronic amateurs and ECE students who want to do quick measurements at home. The oscilloscope proves that our interface is powerful, and it also helps to find out the issues in our interface and I was able to correct some of them and make the interface more robust.

8. Acknowledgment

I'd like to thank Professor Bruce Land first for his idea of making an oscilloscope using this interface as well as his help on debugging some of the hard issues with our interface. He provided many insights on this project and how to make it better. I also want to thank Vipin and Ye for continuing carrying this project while I was gone for one semester. Since there has been a lot of changes since I was gone, Vipin helped clarify some of my questions on the hardware and the software. I want to thank Professor Joseph Skovria for his help on my raspberry Pi related questions as well as giving suggestions on improving the application.

9. Reference

1. ECE 4760 Course Website

<http://people.ece.cornell.edu/land/courses/ece4760/>

2. PIC32 Peripheral Library Guide

<http://ww1.microchip.com/downloads/en/DeviceDoc/32bitPeripheralLibraryGuide.pdf>

3. PIC32 and raspberry Pi of 2018-2019

https://people.ece.cornell.edu/land/courses/eceprojectsland/STUDENTPROJ/2018to2019/vv258_yk749/

4. Raspberry Pi Drawing

<https://learn.adafruit.com/pi-video-output-using-pygame/pygame-drawing-functions>

5. Arcade Game with Raspberry Pi

http://programarcadegames.com/index.php?lang=en&chapter=array_backed_grids

6. Past projects on oscilloscope application

https://courses.ece.cornell.edu/ece5990/ECE5725_Fall2017_projects/th_1630_junpeng_wang/

http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/f2016/ak634_jmw483_dm797/ak634_jmw483_dm797/ak634_jmw483_dm797/index.html

7. PIC ADC - oscilloscope

<https://people.ece.cornell.edu/land/courses/ece4760/PIC32/index.html>

Appendix A. Modified PIC32 Scope Code

```
#ifndef CONFIG_H
#define CONFIG_H
#define _SUPPRESS_PLIB_WARNING
#define _DISABLE_OPENADC10_CONFIGPORT_WARNING
#include <plib.h>
// serial stuff
#include <stdio.h>
#include <string.h>
#include "port_expanderpicandpi.h"

//=====
// 60 MHz
#pragma config FNOSC = FRCPLL, POSCMOD = OFF
#pragma config FPLLIDIV = DIV_2, FPLLMUL = MUL_20, FPLLODIV = DIV_2 //40 MHz
#pragma config FPBDIV = DIV_1 //PB 40 MHz
#pragma config FWDTEN = OFF, JTAGEN = OFF
#pragma config FSOSCEN = OFF //PINS 11 and 12 to secondary oscillator!
#pragma config DEBUG = OFF //RB4 and RB5
//=====
```



```
// set up clock parameters
// system cpu clock
#define sys_clock 40000000

// sys_clock/FPBDIV
#define pb_clock sys_clock/1 // divide by one in this case

#endif /* CONFIG_H */

////////////////////////////////////
// graphics libraries
// SPI channel 1 connections to TFT
#include "config_1_3_2.h"
// threading library
#include "pt_cornell_1_3_2.h"
// yup, the expander
#include "port_expander_br14.h"

#include "tft_master.h"
#include "tft_gfx.h"
// need for rand function
#include <stdlib.h>
// need for sin function
#include <math.h>

//*****define the frequencies and declare variables*****
#define SYS_FREQ 40000000
#define ISR_FREQ 1000

#define DAC_config_chan_A 0b0011000000000000
#define DAC_config_chan_B 0b1011000000000000

char recv_char;

enum SerialState
{ACTIVE_WAIT_FOR_SOT,ACTIVE_PARSE,ACTIVE_CHECKSUM,ACTIVE_WAIT_FOR_EOT};
enum SerialState RecvState;
enum ErrorStatus {ChecksumERR, InvalidCmdERR, NoERR };
enum ErrorStatus RecvStatus;
```

```
#define ComBaudRate 19200
#define DebugBaudRate 9600
#define StartOfTransmit 0xF0
#define EndOfTransmit 0xD7

#define Handshake 0x1A
#define ReadInput 0x2A
#define WriteInput 0x2B
#define DACSetA 0x3A
#define DACSetB 0x3B
#define ConfigDACA 0x3C
#define ConfigDACB 0x3D
#define ConfigDACAB 0x3E
#define StartDAC 0x3F
#define StopDAC 0x39
#define CheckBuf 0x4A
#define SetSampFreq 0x4B
#define StartADC 0x4C
#define SetPWMPer 0x5A
#define StartPWM1 0x5B
#define StartPWM2 0x5C
#define ReadBuf 0x6A
#define WriteBuf 0x6B
#define ByteAck 0x3A

static unsigned char tempbuf[2048];
static unsigned int sys_time_seconds =0;
static char cRecvData[10];
static char cTranData[10];
static char cbuffer[50];
static unsigned char cRecvChar;
static int iChecksumValid=0;
static unsigned char GPIOIN;
static int iPWMScale=0;
//static int timer1_count=0;
int iSampleCh[4],iSampleChBuf[4];
int iTotalNumOfADCSamples=0, iNumOfADCSamples[4];
int iTotalNumOfDACSamples=0;
static unsigned char iBuffer[4][2048];
static signed char iBuffer2[4][2048];
short iDACBuf[2048];
int iDACmode=0;
char ReceiveBuf[256];
```

```
void debug_msg( char* print_buffer){

    int iNumSendChars = 0;
    while (print_buffer[iNumSendChars] != '\0'){
        while(!UARTTransmitterIsReady(UART1));
        UARTSendDataByte(UART1, print_buffer[iNumSendChars]);
        iNumSendChars++;
    }

    iNumSendChars=0;
    while(!UARTTransmitterIsReady(UART1));
    UARTSendDataByte(UART1, '\n');

}

void vExec_Read_Input(char *cRecvData){
    sprintf(cbuffer,"Command %02X Read Input",ReadInput);
    debug_msg(cbuffer);
    GPIOIN = readPE(GPIOZ);
    cTranData[0]= (0xF0 & GPIOIN)>>4;
    cTranData[1]= 0x0F & GPIOIN;

}

void vExec_Write_Input(char *cRecvData){
    char cMSBbits =cRecvData[0];
    char cLSBbits =cRecvData[1];
    // tft_fillScreen(ILI9340_BLACK);
    sprintf(cbuffer,"Command %02X Write Input",WriteInput);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1 %02X Write Bits B7.B6.B5.B4 if 1",cMSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2 %02X Write Bits B3.B2.B1.B0 if 1",cLSBbits);
    debug_msg(cbuffer);
    writePE(GPIOY, cRecvData[0]<<4 | cRecvData[1]);

}

void vExec_DAC_SetA(char *cRecvData){
    char cMSBbits =cRecvData[0];
```

```

    char cLSBbits =cRecvData[1];
    int iDACValue;
    // tft_fillScreen(ILI9340_BLACK);
    sprintf(cbuffer,"Command  %02X  Set DAC Channel A",DACSetA);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1  %02X  DAC MSB Bits B11.B10.B9.B8.B7.B6",cMSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2  %02X  DAC LSB Bits B5.B4.B3.B2.B1.B0",cLSBbits);
    debug_msg(cbuffer);
    iDACValue=((cMSBbits & 0x3f)<<6)|(cLSBbits & 0x3f);
    sprintf(cbuffer,"Set DAC A to %d", iDACValue);
    debug_msg(cbuffer);
    mPORTAClearBits(BIT_3); // start transaction
        delay_ms(100);
    WriteSPI2( DAC_config_chan_A | (iDACValue));
    while (SPI2STATbits.SPIBUSY); // wait for end of transaction
        // CS high
    mPORTASetBits(BIT_3); // end transaction
}

void vExec_DAC_SetB(char *cRecvData){
    char cMSBbits =cRecvData[0];
    char cLSBbits =cRecvData[1];
    int iDACValue;
    // tft_fillScreen(ILI9340_BLACK);
    sprintf(cbuffer,"Command  %02X  Set DAC Channel B",DACSetB);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1  %02X  DAC MSB Bits B11.B10.B9.B8.B7.B6",cMSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2  %02X  DAC LSB Bits B5.B4.B3.B2.B1.B0",cLSBbits);
    debug_msg(cbuffer);
    iDACValue=((cMSBbits & 0x3f)<<6)|(cLSBbits & 0x3f);
    sprintf(cbuffer,"Set DAC B to %d", iDACValue);
    debug_msg(cbuffer);
    mPORTAClearBits(BIT_3); // start transaction

    delay_ms(100);
    WriteSPI2( DAC_config_chan_B | (iDACValue));
    while (SPI2STATbits.SPIBUSY); // wait for end of transaction
        // CS high
    mPORTASetBits(BIT_3); // end transaction
}

```

```

void vExec_Check_Buf(){

    cTranData[0]=
(!iSampleCh[0])+(!iSampleCh[1]<<1)+(!iSampleCh[2]<<2)+(!iSampleCh[3]<<3);
    while(!UARTTransmitterIsReady(UART2));
    UARTSendDataByte(UART2, cTranData[0]);

}

void vExec_Set_Samp_Freq(char *cRecvData){
    unsigned char cSampleFreq= cRecvData[0];
    unsigned char cSampleMSBbits =cRecvData[1];
    unsigned char cSampleLSBbits =cRecvData[2];
    /*sprintf(cbuffer,"Command %02X Set Sample Frequency",SetSampFreq);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1 %02X Sample Frequency %d Khz",cSampleFreq,cSampleFreq);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2 %02X Num of Samples MSB Bits B11.B10.B9.B8.B7.B6",cSampleMSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte3 %02X Num of Samples LSB Bits B5.B4.B3.B2.B1.B0",cSampleLSBbits);
    debug_msg(cbuffer); */
    iTotNumOfADCSamples=((cSampleMSBbits & 0x3f)<<6)|(cSampleLSBbits & 0x3f);
    // sprintf(cbuffer,"Set %dKhz Sample frequency and acquire %d samples",
    cSampleFreq,iTotNumOfADCSamples);
    // debug_msg(cbuffer);
    int timer_count=(sys_clock/(1000*cSampleFreq));
    OpenTimer1(T1_ON | T1_SOURCE_INT | T1_PS_1_1,timer_count );
    ConfigIntTimer1(T1_INT_ON | T1_INT_PRIOR_2);
    mT1ClearIntFlag(); // and clear the interrupt flag

}

void vExec_Start_ADC(char *cRecvData){
    char cChannel= cRecvData[0]>>2;
    char cBuffer=cRecvData[0]&0x03;
    // sprintf(cbuffer,"Command %02X Set Start ADC",StartADC);
    // debug_msg(cbuffer);
    // sprintf(cbuffer,"Byte1 %02X Channel Number %d Buffer Numer
    %d",cRecvData[0],cChannel,cBuffer);
    // debug_msg(cbuffer);

```

```

iSampleCh[cChannel]=1;
iSampleChBuf[cChannel]=cBuffer;
iNumOfADCSamples[cChannel]=0;

}

void vExec_Read_Buf(char *cRecvData){
    char cBufferNum= cRecvData[0];
    char cSampleMSBbits =cRecvData[1];
    char cSampleLSBbits =cRecvData[2];
    int iNumOfSamples;
    int iByteCount=0;
    // printf(cbuffer,"Command %02X Read Buffer",ReadBuf);
    // debug_msg(cbuffer);
    // printf(cbuffer,"Byte1 %02X Buffer Number %d",cBufferNum,cBufferNum);
    // debug_msg(cbuffer);
    // printf(cbuffer,"Byte2 %02X Num of Samples MSB Bits B11.B10.B9.B8.B7.B6",cSampleMSBbits);
    // debug_msg(cbuffer);
    // printf(cbuffer,"Byte3 %02X Num of Samples LSB Bits B5.B4.B3.B2.B1.B0",cSampleLSBbits);
    // debug_msg(cbuffer);
    iNumOfSamples=((cSampleMSBbits<<5)|(cSampleLSBbits & 0x1f));
    // printf(cbuffer,"cSampleMSBbits %d cSampleLSBbits %d ", cSampleMSBbits, );
    // debug_msg(cbuffer);
    // printf(cbuffer,"Read %d Samples from Buffer Number %d", iNumOfSamples,cBufferNum);
    // debug_msg(cbuffer);

    for(iByteCount=0;iByteCount<iNumOfSamples;iByteCount++){
        while(!UARTTransmitterIsReady(UART2));
        UARTSendDataByte(UART2, iBuffer[cBufferNum][iByteCount]);

    }
}

void vExec_Write_Buf(char *cRecvData){
    char cBufferNum= cRecvData[0];
    char cSampleMSBbits =cRecvData[1];
    char cSampleLSBbits =cRecvData[2];
    int iNumOfSamples;
    int iByteCount=0;
    // tft_fillScreen(ILI9340_BLACK);

```

```

sprintf(cbuffer,"Command %02X Write Buffer",WriteBuf);
debug_msg(cbuffer);
sprintf(cbuffer,"Byte1 %02X Buffer Number %d",cBufferNum,cBufferNum);
debug_msg(cbuffer);
sprintf(cbuffer,"Byte2 %02X Num of Samples MSB Bits B11.B10.B9.B8.B7.B6",cSampleMSBbits);
debug_msg(cbuffer);
sprintf(cbuffer,"Byte3 %02X Num of Samples LSB Bits B5.B4.B3.B2.B1.B0",cSampleLSBbits);
debug_msg(cbuffer);
iNumOfSamples=((cSampleMSBbits )<<5)|(cSampleLSBbits & 0x1f);
sprintf(cbuffer,"Write %d Samples to Buffer Number %d", iNumOfSamples,cBufferNum);
debug_msg(cbuffer);

for(iByteCount=0;iByteCount<iNumOfSamples;iByteCount++){
    while(!UARTReceivedDataIsAvailable(UART2));
    iBuffer[cBufferNum][iByteCount]=UARTGetDataByte(UART2);
    if((iByteCount%8)==0)
        UARTSendDataByte(UART2, ByteAck);
}

}

void vExec_Set_PWM_Per(char *cRecvData){
    char cPeriodMSBbits= cRecvData[0];
    char cPeriodLSBbits =cRecvData[1];
    char cPeriodunit =cRecvData[2];

    int iPeriod;
    // tft_fillScreen(ILI9340_BLACK);
    sprintf(cbuffer,"Command %02X Set PWM Period",SetPWMPer);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1 %02X Period MSB B11.B10.B9.B8.B7.B6",cPeriodMSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2 %02X Period LSB Bits B5.B4.B3.B2.B1.B0",cPeriodLSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte3 %02X Unit",cPeriodunit);
    debug_msg(cbuffer);
    iPeriod=(((cPeriodMSBbits & 0x1f)<<5)|cPeriodLSBbits);
    iPeriod=iPeriod*(pow(10,cPeriodunit));
    sprintf(cbuffer,"Set PWM Period to %d us",iPeriod);
    debug_msg(cbuffer);
    // delay_ms(100);

```

```

CloseTimer2();
if((iPeriod*40)<pow(2,16)){
OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, iPeriod*40);
iPWMScale=0;
}

else if((iPeriod*40)<pow(2,17)){
OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_2, (iPeriod*40)>>1);
iPWMScale=1;
}
else if((iPeriod*40)<pow(2,18)){
OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_4, (iPeriod*40)>>2);
iPWMScale=2;
}
else if((iPeriod*40)<pow(2,19)){
OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_8, (iPeriod*40)>>3);
iPWMScale=3;
}
else if((iPeriod*40)<pow(2,20)){
OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_16, (iPeriod*40)>>4);
iPWMScale=4;
}
OpenOC1(OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE , 0, iPeriod); //
OpenOC4(OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE , 0, iPeriod); //

}

void vExec_Start_PWM1(char *cRecvData){
char cOnTime1MSBbits= cRecvData[0];
char cOnTime1LSBbits =cRecvData[1];
char cPeriodunit =cRecvData[2];

int iPWM1ONPeriod;
// tft_fillScreen(ILI9340_BLACK);
printf(cbuffer,"Command %02X Set PWM 1",StartPWM1);
debug_msg(cbuffer);
printf(cbuffer,"Byte1 %02X On Time 1 MSB B11.B10.B9.B8.B7.B6",cOnTime1MSBbits);
debug_msg(cbuffer);
printf(cbuffer,"Byte2 %02X On Time 1 LSB Bits B5.B4.B3.B2.B1.B0",cOnTime1LSBbits);
debug_msg(cbuffer);

```



```
    sprintf(cbuffer,"Byte3 %02X Unit ",cPeriodunit);
    debug_msg(cbuffer);
    iPWM1ONPeriod=(((cOnTime1MSBbits & 0x1f)<<5)|cOnTime1LSBbits);
    iPWM1ONPeriod=((iPWM1ONPeriod*40)>>iPWMScale)*pow(10,cPeriodunit);
    sprintf(cbuffer,"Set PWM Period to %d us",iPWM1ONPeriod);
    debug_msg(cbuffer);
    SetDCOC1PWM(iPWM1ONPeriod);

}

void vExec_Start_PWM2(char *cRecvData){
    char cOnTime2MSBbits= cRecvData[0];
    char cOnTime2LSBbits =cRecvData[1];
    char cPeriodunit =cRecvData[2];
    int iPWM2ONPeriod;
    // tft_fillScreen(ILI9340_BLACK);
    sprintf(cbuffer,"Command %02X Set PWM 2",StartPWM2);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1 %02X On Time 1 MSB B11.B10.B9.B8.B7.B6",cOnTime2MSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2 %02X On Time 1 LSB B5.B4.B3.B2.B1.B0",cOnTime2LSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte3 %02X Unit",cPeriodunit);
    debug_msg(cbuffer);
    iPWM2ONPeriod=(((cOnTime2MSBbits & 0x1f)<<5)|cOnTime2LSBbits);
    iPWM2ONPeriod=((iPWM2ONPeriod*40)>>iPWMScale)*pow(10,cPeriodunit);
    sprintf(cbuffer,"Set PWM Period to %d us",iPWM2ONPeriod);
    debug_msg(cbuffer);
    SetDCOC4PWM(iPWM2ONPeriod);
}

void vExec_DAC_ConfigA(char *cRecvData){
    char cBufferNum= cRecvData[0];
    iDACmode=cRecvData[1];
    sprintf(cbuffer,"Command %02X Set Config DACA",ConfigDACA);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1 %02X Source Buffer Number %d",cRecvData[0],cBufferNum);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2 %02X DAC mode %d",cRecvData[1],cRecvData[1]);
    debug_msg(cbuffer);
    int *pDAC1= &(iBuffer[cBufferNum][0]);
```

```

    int i=0;
    CloseTimer3();
    DmaChnDisable(DMA_CHANNEL3);
    for(i=0;i<1024;i++){
        iDACBuf[i]=DAC_config_chan_A*pDAC1;
        pDAC1++;
    }
}

void vExec_DAC_ConfigB(char *cRecvData){
    char cBufferNum= cRecvData[0];
    iDACmode=cRecvData[1];
    sprintf(cbuffer,"Command  %02X  Set Config DACB",ConfigDACB);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1  %02X  Source Buffer Number %d",cRecvData[0],cBufferNum);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2  %02X  DAC mode %d",cRecvData[1],cRecvData[1]);
    debug_msg(cbuffer);
    CloseTimer3();
    DmaChnDisable(DMA_CHANNEL3);
    int *pDAC2= &(iBuffer[cBufferNum][0]);
    int i=0;
    for(i=0;i<1024;i++){
        iDACBuf[i]=DAC_config_chan_B*pDAC2;
        pDAC2++;
    }
}

void vExec_DAC_ConfigAB(char *cRecvData){
    char cBufferA= cRecvData[0]>>2;
    char cBufferB=cRecvData[0]&0x3;
    iDACmode=cRecvData[1]+2;
    sprintf(cbuffer,"Command  %02X  Set Config DAC A and B",ConfigDACAB);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1  %02X  Source Buffer NumberA %d Buffer NumberB
%d",cRecvData[0],cBufferA,cBufferB);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2  %02X  DAC mode %d",cRecvData[1],cRecvData[1]);
    debug_msg(cbuffer);
    CloseTimer3();
    DmaChnDisable(DMA_CHANNEL3);
    short *pDAC1= &(iBuffer[cBufferA][0]);

```

```

short *pDAC2= &(iBuffer[cBufferB][0]);
int i=0;
short *pDAC3;
pDAC3=pDAC1;

for(i=0;i<2048;i+=2){
    iDACBuf[i]=DAC_config_chan_A/*pDAC1;
    iDACBuf[i+1]=DAC_config_chan_B/*pDAC2;
    pDAC1++;
    pDAC2++;

}

}

void vExec_DAC_Start(char *cRecvData){
    char cSampleFreq= cRecvData[0];
    char cSampleMSBbits =cRecvData[1];
    char cSampleLSBbits =cRecvData[2];
    // tft_fillScreen(ILI9340_BLACK);
    sprintf(cbuffer,"Command %02X Set Sample Frequency",StartDAC);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte1 %02X Sample Frequency %d Khz",cSampleFreq,cSampleFreq);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte2 %02X Num of Samples MSB Bits B11.B10.B9.B8.B7.B6",cSampleMSBbits);
    debug_msg(cbuffer);
    sprintf(cbuffer,"Byte3 %02X Num of Samples LSB Bits B5.B4.B3.B2.B1.B0",cSampleLSBbits);
    debug_msg(cbuffer);
    iTotNumOfDACSamples=((cSampleMSBbits & 0x3f)<<5)|(cSampleLSBbits & 0x1f);
    sprintf(cbuffer,"Set %dKhz Sample frequency and generate %d samples",
cSampleFreq,iTotNumOfDACSamples);
    debug_msg(cbuffer);

    switch(iDACmode){
        case 0:  OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1,
SYS_FREQ/(cSampleFreq<<10));
                DmaChnOpen(DMA_CHANNEL3, 0, DMA_OPEN_DEFAULT);
                DmaChnSetTxfer(DMA_CHANNEL3,(short *) iDACBuf, &SPI2BUF,
iTotNumOfDACSamples*2, 2, 2 );

                break;
        case 1:  OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1,SYS_FREQ/(cSampleFreq<<10)
);

```

```
        DmaChnOpen(DMA_CHANNEL3, 0, DMA_OPEN_AUTO);
        DmaChnSetTxfer(DMA_CHANNEL3, (short *)iDACBuf, &SPI2BUF,
iTotalNumOfDACSamples*2, 2, 2 );
        break;
    case 2:   OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1, SYS_FREQ/(cSampleFreq<<11)
);
        DmaChnOpen(DMA_CHANNEL3, 0, DMA_OPEN_DEFAULT);
        DmaChnSetTxfer(DMA_CHANNEL3, (short *)iDACBuf, &SPI2BUF,
iTotalNumOfDACSamples*4, 2, 2 );
        break;
    case 3:   OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1,
SYS_FREQ/(cSampleFreq<<11));
        DmaChnOpen(DMA_CHANNEL3, 0, DMA_OPEN_AUTO);
        DmaChnSetTxfer(DMA_CHANNEL3, (short *)iDACBuf, &SPI2BUF,
iTotalNumOfDACSamples*4, 2, 2 );
        break;
    }

    DmaChnSetEventControl(DMA_CHANNEL3, DMA_EV_START_IRQ(_TIMER_3_IRQ));
    DmaChnEnable(DMA_CHANNEL3);

}

void vExec_DAC_Stop(){
    sprintf(cbuffer,"Command  %02X  Stop DAC",StopDAC);
    debug_msg(cbuffer);
    CloseTimer3();
    DmaChnDisable(DMA_CHANNEL3);
}

int CheckSum(char *cRecvData, int iNumOfBytes, char iSum){

    return 1;

}

void vSendErrorMsg(char RecvStatus){

}
```

```
int iNumOfDataBytes(unsigned char cRecvChar){
    switch(cRecvChar){

        case(ReadInput):    return 0;

        case(WriteInput):   return 2;

        case(DACSetA):      return 2;

        case(DACSetB):      return 2;

        case(ConfigDACA):   return 2;

        case(ConfigDACB):   return 2;

        case(ConfigDACAB):  return 2;

        case(StartDAC):     return 3;

        case(StopDAC):      return 0;

        case(CheckBuf):     return 0;

        case(SetSampFreq):  return 3;

        case(StartADC):     return 1;

        case(SetPWMPer):    return 3;

        case(StartPWM1):    return 3;

        case(StartPWM2):    return 3;

        case(ReadBuf):      return 3;

        case(WriteBuf):     return 3;

        default:            return -1;
    }
}
```

```
int iNumOfRespBytes(unsigned char cRecvChar){
    switch(cRecvChar){

        case(ReadInput):    return 2;

        case(WriteInput):   return 0;

        case(DACSetA):      return 0;

        case(DACSetB):      return 0;

        case(ConfigDACA):   return 0;

        case(ConfigDACB):   return 0;

        case(ConfigDACAB):  return 0;

        case(StartDAC):     return 0;

        case(StopDAC):      return 0;

        case(CheckBuf):     return 1;

        case(SetSampFreq):  return 0;

        case(StartADC):     return 0;

        case(SetPWMPer):    return 0;

        case(StartPWM1):    return 0;

        case(StartPWM2):    return 0;

        case(ReadBuf):      return 0;

        case(WriteBuf):     return 0;

        default:            return -1;
    }
}
```

```
void vSendRespMsg(unsigned char cCommand,char *cTansData)
{
    while(!UARTTransmitterIsReady(UART2));
    UARTSendDataByte(UART2,StartOfTransmit);

    while(!UARTTransmitterIsReady(UART2));
    UARTSendDataByte(UART2,cCommand);
    int iByte,iRespBytes;
    iRespBytes=iNumOfRespBytes(cCommand);
    for(iByte=0;iByte<iRespBytes;iByte++){
        while(!UARTTransmitterIsReady(UART2));
        UARTSendDataByte(UART2,cTansData[iByte]);
    }

    while(!UARTTransmitterIsReady(UART2));
    UARTSendDataByte(UART2,EndOfTransmit);
}
void vExecuteCommand(unsigned char cCommand,char *cRecvData){
    switch(cCommand){

        case(ReadInput):  vExec_Read_Input(cRecvData);  break;

        case(WriteInput): vExec_Write_Input(cRecvData); break;

        case(DACSetA):    vExec_DAC_SetA(cRecvData);    break;

        case(DACSetB):    vExec_DAC_SetB(cRecvData);    break;

        case(ConfigDACA): vExec_DAC_ConfigA(cRecvData); break;

        case(ConfigDACB): vExec_DAC_ConfigB(cRecvData); break;

        case(ConfigDACAB): vExec_DAC_ConfigAB(cRecvData); break;

        case(StartDAC):   vExec_DAC_Start(cRecvData);   break;

        case(StopDAC):    vExec_DAC_Stop(cRecvData);    break;

        case(CheckBuf):   vExec_Check_Buf();             break;

        case(SetSampFreq): vExec_Set_Samp_Freq(cRecvData); break;
    }
}
```

```
        case(StartADC):    vExec_Start_ADC(cRecvData);    break;

        case(SetPWMPer):   vExec_Set_PWM_Per(cRecvData);  break;

        case(StartPWM1):   vExec_Start_PWM1(cRecvData);  break;

        case(StartPWM2):   vExec_Start_PWM2(cRecvData);  break;

        case(ReadBuf):     vExec_Read_Buf(cRecvData);     break;

        case(WriteBuf):    vExec_Write_Buf(cRecvData);    break;

    }

}

void __ISR(_TIMER_1_VECTOR, IPL3) Timer1Handler(void){
mT1ClearIntFlag();
mPORTBToggleBits(BIT_4);//
//AcquireADC10();
static int iChannelCount, iSampleCount;

for(iChannelCount=0;iChannelCount<4;iChannelCount++){
    if(iNumOfADCSamples[iChannelCount]==iTotalNumOfADCSamples){
        iSampleCh[iChannelCount]=0;
    }else if(iSampleCh[iChannelCount]){
        iSampleCount=(iNumOfADCSamples[iChannelCount])++;
        short *buffer_ptr=(short *) (iBuffer[iSampleChBuf[iChannelCount]]);
        unsigned short adc_val=ReadADC10(iChannelCount);
        *(buffer_ptr+iSampleCount)=adc_val;

    }

}

}
```



```
static struct pt pt_uart_receive, pt_timer;
```

```
static int iNumOfRecvBytes=0;
```

```
static int iByteCount=0;
```

```
static unsigned char cCommand;
```

```
static PT_THREAD (protothread_timer(struct pt *pt))
```

```
{
```

```
    PT_BEGIN(pt);
```

```
        PT_YIELD_TIME_msec(1000) ;
```

```
        sys_time_seconds++ ;
```

```
    PT_END(pt);
```

```
}
```

```
void PIN_MANAGER_Initialize(void)
```

```
{
```

```
//PPSInput( 3 , IC1 , RPA2 );
```

```
PPSInput( 3 , U1RX , RPA4 );
```

```
PPSInput( 2 , U2RX , RPB5 );
```

```
//PPSInput( 3 , SDI2 , RPB13 );
```

```
//PPSInput( 4 , IC2 , RPA3 );
```

```
PPSOutput( 1 , RPB7 , U1TX );
```

```
PPSOutput( 1 , RPB3 , OC1 );
```

```
PPSOutput( 4 , RPB9 , SS2 );
```

```
PPSOutput( 4 , RPB10 , U2TX );
```

```
PPSOutput( 3 , RPB2 , OC4 );
```

```
PPSOutput( 2 , RPB11 , SDO2 );
```

```
}
```

```
void Variable_Initialize(void) {
```

```
    iSampleCh[0]=0;
```

```
    iSampleCh[1]=0;
```

```
    iSampleCh[2]=0;
```

```
    iSampleCh[3]=0;
```

```
    iNumOfADCSamples[0]=0;
```

```
    iNumOfADCSamples[1]=0;
```

```
    iNumOfADCSamples[2]=0;
```

```
iNumOfADCSamples[3]=0;
RecvState= ACTIVE_WAIT_FOR_SOT;

}

// handler for the DMA channel 1 interrupt
void __ISR(_DMA1_VECTOR, IPL5SOFT) DmaHandler1(void)
{
    DmaChnDisable(DMA_CHANNEL1);
    INTClearFlag(INT_SOURCE_DMA(DMA_CHANNEL1));    // release the interrupt in the
    INT controller, we're servicing int

    int iByte=0;

    do{
        cRecvChar=ReceiveBuf[iByte++];
        if(RecvState==ACTIVE_WAIT_FOR_SOT && cRecvChar== 0xF0){
            RecvState=ACTIVE_PARSE;
            iChecksumValid=0;
            RecvStatus=NoERR;
        }

        else if (RecvState==ACTIVE_WAIT_FOR_EOT && cRecvChar== EndOfTransmit){
            if(RecvStatus==NoERR){
                if(cCommand==ReadBuf || cCommand==WriteBuf){
                    vSendRespMsg(cCommand,cTranData);
                    vExecuteCommand(cCommand,cRecvData);
                }
                else{
                    vExecuteCommand(cCommand,cRecvData);
                    vSendRespMsg(cCommand,cTranData);
                }
            }
            else
                vSendErrorMsg(RecvStatus);

            RecvState=ACTIVE_WAIT_FOR_SOT;
        }

        else if (RecvState==ACTIVE_CHECKSUM){
            if(!Checksum(cRecvData,iNumOfRecvBytes,cRecvChar))
                RecvStatus=ChecksumERR;
        }
    }
}
```

```

    RecvState=ACTIVE_WAIT_FOR_EOT;
    }

    else if(RcvState==ACTIVE_PARSE){
        cCommand=cRecvChar;
        iNumOfRecvBytes=iNumOfDataBytes(cCommand);

        if(iNumOfRecvBytes ==-1)
            RecvStatus=InvalidCmdERR;

        else{

            for(iByteCount=0;iByteCount<iNumOfRecvBytes;iByteCount++){
                cRecvData[iByteCount]=ReceiveBuf[iByte++];
            }
        }
        RecvState=ACTIVE_CHECKSUM;
    }

} while(cRecvChar!= EndOfTransmit);

    DmaChnClrEvFlags(DMA_CHANNEL1, DMA_EV_BLOCK_DONE);
    INTEnable(INT_SOURCE_DMA(DMA_CHANNEL1), INT_ENABLED);           // enable the
    chn interrupt in the INT controller

    // enable the chn
    DmaChnEnable(DMA_CHANNEL1);
}

void main(){

    PT_setup();
    PIN_MANAGER_Initialize();
    Variable_Initialize();

    // === setup system wide interrupts =====
    INTEnableSystemMultiVectoredInt();

    //====setup UART 1 to debug=====

    UARTConfigure(UART1, UART_ENABLE_PINS_TX_RX_ONLY);

```

```

UARTSetLineControl(UART1, UART_DATA_SIZE_8_BITS | UART_PARITY_NONE |
UART_STOP_BITS_1);
UARTSetDataRate(UART1, pb_clock, DebugBaudRate);
UARTEnable(UART1, UART_ENABLE_FLAGS(UART_PERIPHERAL | UART_RX | UART_TX));

```

```

//=====setup UART 2 to communicate with RPi=====

```

```

UARTConfigure(UART2, UART_ENABLE_PINS_TX_RX_ONLY);
UARTSetLineControl(UART2, UART_DATA_SIZE_8_BITS | UART_PARITY_NONE |
UART_STOP_BITS_1);
UARTSetDataRate(UART2, pb_clock, ComBaudRate);
UARTEnable(UART2, UART_ENABLE_FLAGS(UART_PERIPHERAL | UART_RX | UART_TX));

```

```

ANSELA =0;
ANSELB =0;
// the ADC //////////////////////////////////////

```

```

// ---- configure and enable the ADC ----

```

```

// ensure the ADC is off before setting the configuration
CloseADC10();

```

```

// define setup parameters for OpenADC10
// Turn module on | output in integer | trigger mode auto | enable autosample
// ADC_CLK_AUTO -- Internal counter ends sampling and starts conversion (Auto convert)
// ADC_CLK_TMR -- triggered off timer3 match
// ADC_AUTO_SAMPLING_ON -- Sampling begins immediately after last conversion completes;
SAMP bit is automatically set
// ADC_AUTO_SAMPLING_OFF -- Sampling begins with AcquireADC10();
#define PARAM1 ADC_FORMAT_INTG32 | ADC_CLK_AUTO | ADC_AUTO_SAMPLING_ON
//ADC_CLK_TMR ADC_CLK_AUTO

```

```

// define setup parameters for OpenADC10
// ADC ref external | disable offset test | disable scan mode | do 1 sample | use single buf | alternate
mode off
#define PARAM2 ADC_VREF_AVDD_AVSS | ADC_OFFSET_CAL_DISABLE |
ADC_SCAN_OFF | ADC_SAMPLES_PER_INT_1 | ADC_ALT_BUF_OFF | ADC_ALT_INPUT_OFF
//
// Define setup parameters for OpenADC10
// use peripheral bus clock | set sample time | set ADC clock divider
// ADC_CONV_CLK_Tcy2 means divide CLK_PB by 2 (max speed)
// ADC_SAMPLE_TIME_5 seems to work with a source resistance < 1kohm

```

```
// At PB clock 30 MHz, divide by two for ADC_CONV_CLK gives 66 nSec
#define PARAM3 ADC_CONV_CLK_INTERNAL_RC | ADC_SAMPLE_TIME_15

// define setup parameters for OpenADC10
// set AN0 and as analog inputs
#define PARAM4 ENABLE_AN0_ANA

// define setup parameters for OpenADC10
// do not assign channels to scan
#define PARAM5 SKIP_SCAN_AN1 |SKIP_SCAN_AN2|SKIP_SCAN_AN3|SKIP_SCAN_AN4
|SKIP_SCAN_AN5 |SKIP_SCAN_AN6 |SKIP_SCAN_AN7 | \
SKIP_SCAN_AN8 |SKIP_SCAN_AN9 |SKIP_SCAN_AN10 |SKIP_SCAN_AN11 | \
SKIP_SCAN_AN12 |SKIP_SCAN_AN13 |SKIP_SCAN_AN14 |SKIP_SCAN_AN15

// use ground as neg ref for A | use AN0 for input A
// configure to sample AN4
SetChanADC10( ADC_CH0_NEG_SAMPLEA_NVREF ); // configure to sample AN0
OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5 ); // configure ADC using the
parameters defined above

EnableADC10(); // Enable the ADC
SpiChnOpen(SPI_CHANNEL2, SPI_OPEN_ON | SPI_OPEN_MODE16 | SPI_OPEN_MSTEN |
SPI_OPEN_CKE_REV | SPICON_FRMEN | SPICON_FRMPOL, 4);

//GPIO INIT
mPORTBSetPinsDigitalOut(BIT_4);
mPORTBSetBits(BIT_4);

//Port Expander
initPE();
// Outputs
mPortYSetPinsOut(BIT_7 | BIT_6 | BIT_5 | BIT_4 |BIT_3 | BIT_2 |BIT_1 | BIT_0 );
// Inputs
mPortZSetPinsIn(BIT_7 | BIT_6 | BIT_5 | BIT_4 |BIT_3 | BIT_2 |BIT_1 | BIT_0 );
// Input pull up resistors
mPortZEnablePullUp(BIT_7 | BIT_6 | BIT_5 | BIT_4 |BIT_3 | BIT_2 |BIT_1 | BIT_0 );

PT_INIT(&pt_timer);
PT_INIT(&pt_uart_receive);
sprintf(cbuffer,"Welcome to PIC & Pi Project Debug Window\n");
```

```

    debug_msg(cbuffer);
char print_buffer[50]="Welcome to PIC & Pi Project Command Window\n";
int iNumSendChars = 0;
while (print_buffer[iNumSendChars] != '\0'){
    while(!UARTTransmitterIsReady(UART2));
    UARTSendDataByte(UART2, print_buffer[iNumSendChars]);
    iNumSendChars++;
}

iNumSendChars=0;
while(!UARTTransmitterIsReady(UART2));
    UARTSendDataByte(UART2, '\n');

DmaChnOpen(DMA_CHANNEL1, DMA_CHN_PRI2, DMA_OPEN_MATCH);

    DmaChnSetMatchPattern(DMA_CHANNEL1, EndOfTransmit);// set \r as ending character

    // set the events: we want the UART2 rx interrupt to start our transfer
    // also we want to enable the pattern match: transfer stops upon detection of EOT
    DmaChnSetEventControl(DMA_CHANNEL1,
DMA_EV_START_IRQ_EN|DMA_EV_MATCH_EN|DMA_EV_START_IRQ(_UART2_RX_IRQ));

    // set the transfer source and dest addresses, source and dest sizes and the cell size
    DmaChnSetTxfer(DMA_CHANNEL1, (void*)&U2RXREG, ReceiveBuf, 1, 256, 1);

    DmaChnSetEvEnableFlags(DMA_CHANNEL1, DMA_EV_BLOCK_DONE);           //
enable the transfer done interrupt: pattern match or all the characters transferred

    // enable system wide multi vectored interrupts
    INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
    INTEnableInterrupts();

    INTSetVectorPriority(INT_VECTOR_DMA(DMA_CHANNEL1),
INT_PRIORITY_LEVEL_5);           // set INT controller priority
    INTSetVectorSubPriority(INT_VECTOR_DMA(DMA_CHANNEL1),
INT_SUB_PRIORITY_LEVEL_3);       // set INT controller sub-priority

    INTEnable(INT_SOURCE_DMA(DMA_CHANNEL1), INT_ENABLED);           //
enable the chn interrupt in the INT controller

    // enable the chn
    DmaChnEnable(DMA_CHANNEL1);

```

```
while (1){  
    PT_SCHEDULE(protothread_timer(&pt_timer));  
  
}  
  
}
```

Appendix B. Modified Raspberry Pi Interface Code

```
import serial  
import binascii  
import time  
  
ComBaudRate    = 115200  
StartOfTransmit = 0xF0  
EndOfTransmit  = 0xD7  
  
Handshake      = 0x1A  
ReadInput      = 0x2A  
WriteInput     = 0x2B  
DACSetA        = 0x3A  
DACSetB        = 0x3B  
CheckBuf       = 0x4A  
SetSampFreq    = 0x4B  
StartADC       = 0x4C  
SetPWMPer      = 0x5A  
StartPWM1      = 0x5B  
StartPWM2      = 0x5C  
ReadBuf        = 0x6A  
WriteBuf       = 0x6B  
ConfigDACA     = 0x3C  
ConfigDACB     = 0x3D  
ConfigDACAB    = 0x3E  
StartDAC       = 0x3F  
StopDAC        = 0x39
```

```
ser = serial.Serial( "/dev/serial0",
    baudrate=ComBaudRate,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    #timeout=5, # IMPORTANT, can be lower or higher
)
def CheckSum(Command):
    CheckSumValue=0x0
    for i in range(0,len(Command)):
        CheckSumValue=CheckSumValue+Command[i]
    return CheckSumValue

def SendCommand(Command):
    TransArray=[StartOfTransmit]
    CmdLength=len(Command)
    for i in range(0,CmdLength):
        TransArray.append(Command[i])
    TransArray.append(CheckSum(Command))
    TransArray.append(EndOfTransmit)
    for cmd_byte in TransArray:
        hex_byte = ("{:02x}".format(cmd_byte))
        # print(hex_byte)
        ser.write(bytearray.fromhex(hex_byte))

def WriteBuffer(BufNum, Data):
    DataLength=len(Data)
    DataLengthMSB,DataLengthLSB=divmod(DataLength,32)
    Command = [WriteBuf,BufNum,DataLengthMSB,DataLengthLSB]
    SendCommand(Command)
    #time.sleep(0.1)
    data_bytes=ser.read(3)

    for i in range(0,DataLength):
        hex_byte = ("{:02x}".format(Data[i]))
        # time.sleep(0.04)
        # print(hex_byte)
        ser.write(bytearray.fromhex(hex_byte))
        if((i%8)==0):
            ser.read(1)
```



```
def ReadBuffer(BufNum, DataLength):
    DataLengthMSB,DataLengthLSB=divmod(DataLength,32)
    Command = [ReadBuf,BufNum,DataLengthMSB,DataLengthLSB]
    Data = list()
    SendCommand(Command)
    data_bytes=ser.read(3)

    data_bytes=ser.read(DataLength)
    data_bytes_hex=data_bytes.encode('hex')
    resp_bytes = bytearray.fromhex(data_bytes_hex)
    for data_byte in resp_bytes:
        Data.append(data_byte)

    return Data

def ReadBuffer2(BufNum, DataLength):
    DataLengthMSB,DataLengthLSB=divmod(2*DataLength,32)
    Command = [ReadBuf,BufNum,DataLengthMSB,DataLengthLSB]
    Data = list()
    Data2 = list()

    SendCommand(Command)
    data_bytes=ser.read(3)
    data_bytes=ser.read(2*DataLength)
    data_bytes_hex=data_bytes.encode('hex')
    resp_bytes = bytearray.fromhex(data_bytes_hex)
    for data_byte in resp_bytes:
        Data.append(data_byte)
    for i in range(0,DataLength-1):
        Data2.append(Data[2*i]+Data[2*i+1]*256)
    return Data2

def SetPWMPeriod(Period,unit):
    PeriodMSB,PeriodLSB=divmod(Period,32)
    Command =[SetPWMPer,PeriodMSB,PeriodLSB,unit]
    SendCommand(Command)
    data_bytes=ser.read(3)
```

```
def EnablePWM1(OnTime,unit):  
    OnTimeMSB,OnTimeLSB=divmod(OnTime,32)  
    Command =[StartPWM1,OnTimeMSB,OnTimeLSB,unit]  
    SendCommand(Command)  
    data_bytes=ser.read(3)
```

```
def EnablePWM2(OnTime,unit):  
    OnTimeMSB,OnTimeLSB=divmod(OnTime,32)  
    Command =[StartPWM2,OnTimeMSB,OnTimeLSB,unit]  
    SendCommand(Command)  
    data_bytes=ser.read(3)
```

```
def SetDACA(DacVal):  
    DacValMSB,DacValLSB=divmod(DacVal,64)  
    Command =[DACSetA,DacValMSB,DacValLSB]  
    SendCommand(Command)  
    data_bytes=ser.read(3)
```

```
def SetDACB(DacVal):  
    DacValMSB,DacValLSB=divmod(DacVal,64)  
    Command =[DACSetB,DacValMSB,DacValLSB]  
    SendCommand(Command)  
    data_bytes=ser.read(3)
```

```
def ConfigureDACA(BufNum, Mode):  
    Command =[ConfigDACA,BufNum,Mode]  
    SendCommand(Command)  
    data_bytes=ser.read(3)
```

```
def ConfigureDACB(BufNum, Mode):  
    Command =[ConfigDACB,BufNum,Mode]  
    SendCommand(Command)  
    data_bytes=ser.read(3)
```

```
def ConfigureDACAB(BufNumA,BufNumB, Mode):
    BufNum=BufNumA*4+BufNumB
    Command =[ConfigDACAB,BufNum,Mode]
    SendCommand(Command)
    data_bytes=ser.read(3)

def StartDACOutput(SampleFreq, Samples):
    SamplesMSB,SamplesLSB=divmod(Samples,32)
    Command =[StartDAC,SampleFreq,SamplesMSB,SamplesLSB]
    SendCommand(Command)
    data_bytes=ser.read(3)

def StopDACOutput():
    Command =[StopDAC]
    SendCommand(Command)
    data_bytes=ser.read(3)

def CheckBufferStatus():
    Command = [CheckBuf]
    SendCommand(Command)
    print "finsihed send command"
    data_bytes=ser.read(4)
    data_bytes_hex=data_bytes.encode('hex')
    resp_bytes = bytearray.fromhex(data_bytes_hex)
    status=resp_bytes[2];
    print "print status===="
    print status
    print "====="
    return status

def _StartADC_( Channel,Buffer ):
    ChannelAndBuffer = Channel*4 + Buffer
    Command = [StartADC,ChannelAndBuffer]
    SendCommand(Command)
    data_bytes=ser.read(3)
```

```
def _SetSampleFreq_( PrescalerSetting, SampleVal ):
    SampleValMSB,SampleValLSB = divmod( SampleVal, 64 )
    Command = [ SetSampFreq, PrescalerSetting, SampleValMSB,SampleValLSB ]
    SendCommand(Command)
    data_bytes=ser.read(3)
```

```
def WriteGPIO(data):
    DataValMSB,DataValLSB = divmod( data, 16 )
    Command = [ WriteInput, DataValMSB, DataValLSB ]
    SendCommand(Command)
    data_bytes=ser.read(3)
```

```
def ReadGPIO():
    Command = [ ReadInput ]
    SendCommand(Command)
    data_bytes=ser.read(5)
    data_bytes_hex=data_bytes.encode('hex')
    gpio_bytes = bytearray.fromhex(data_bytes_hex)
    ReadVal=gpio_bytes[2]*16+gpio_bytes[3];

    return ReadVal
```

Appendix C. Oscilloscope Code

```
import pygame
import sys
sys.path.append('./')
import PIC32Interface
import time
import os
import RPi.GPIO as GPIO

#display on piTFT
os.putenv('SDL_VIDEORIVER','fbcon')
os.putenv('SDL_FBDEV','/dev/fb1')
```

```
#initialize GPIO
```

```
GPIO.setmode(GPIO.BCM)
```

```
GPIO.setup(4, GPIO.IN, pull_up_down=GPIO.PUD_UP)  
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)  
GPIO.setup(22, GPIO.IN, pull_up_down=GPIO.PUD_UP)  
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)  
GPIO.setup(27, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

```
#callback functions
```

```
#time Up
```

```
def GPIO17_callback(channel):
```

```
    global time_Index
```

```
    time_Index= time_Index+1
```

```
    if (time_Index> 5):
```

```
        time_Index= 5
```

```
    print(time_Index)
```

```
#time Down
```

```
def GPIO22_callback(channel):
```

```
    global time_Index
```

```
    time_Index= time_Index-1
```

```
    if (time_Index<0):
```

```
        time_Index= 0
```

```
    print(time_Index)
```

```
#volt Up
```

```
def GPIO23_callback(channel):
```

```
    global volt_Index
```

```
    volt_Index = volt_Index+1
```

```
    if (volt_Index> 3):
```

```
        volt_Index= 3
```

```
print(volt_Index)
```

```
#time Down
```

```
def GPIO27_callback(channel):
```

```
    global volt_Index
```

```
    volt_Index = volt_Index-1
```

```
    if (volt_Index < 0):
```

```
        volt_Index = 0
```

```
    print(volt_Index)
```

```
#add event detection for the GPIO
```

```
GPIO.add_event_detect(17, GPIO.FALLING, callback=GPIO17_callback, bouncetime=300)
```

```
GPIO.add_event_detect(22, GPIO.FALLING, callback=GPIO22_callback, bouncetime=300)
```

```
GPIO.add_event_detect(23, GPIO.FALLING, callback=GPIO23_callback, bouncetime=300)
```

```
GPIO.add_event_detect(27, GPIO.FALLING, callback=GPIO27_callback, bouncetime=300)
```

```
# Define some colors
```

```
BLACK = (0, 0, 0)
```

```
WHITE = (255, 255, 255)
```

```
GREEN = (0, 255, 0)
```

```
RED = (255, 0, 0)
```

```
PURPLE = (238, 130, 238)
```

```
top = 0
```

```
bottom = 239
```

```
left = 0
```

```
right = 319
```

```
tpd_list = ['200us', '500us', ' 1ms', ' 2ms', ' 10ms', '20ms']
```

```
vpd_list = [' 2V', ' 1V', '500mV', '250mV']
```

```
vpd_list_num = [2, 1, 0.5, 0.25]
```

```
sample_list = [160, 64, 32, 16, 6, 3]
```

```
time_Index = 0
```

```
volt_Index = 1
```

```
x_axis = []
```

```
lines = []

# Initialize pygame
pygame.init()

# Set the HEIGHT and WIDTH of the screen
WINDOW_SIZE = [320, 240]
screen = pygame.display.set_mode(WINDOW_SIZE)

# Set title of screen
pygame.display.set_caption("Pic&Pi Scope")

# Loop until the user clicks the close button.
done = False

# Used to manage how fast the screen updates
clock = pygame.time.Clock()

data_len = 255
for i in range(0,data_len):
    lines.append([0,0])
    x_axis.append(i*1.25)

# ----- Main Program Loop -----
while not done:

    if(not GPIO.input(4)) : break;
    for event in pygame.event.get(): # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done = True # Flag that we are done so we exit this loop

    #=====Pic & Pi set up sample frquence and read Data=====
    PIC32Interface._SetSampleFreq_( sample_list[time_Index],512)
    PIC32Interface._StartADC_( 0,3 )
    time.sleep(1)
    while((PIC32Interface.CheckBufferStatus() != 0x0F) ):
        time.sleep(0.05)
    ReadData=PIC32Interface.ReadBuffer2(3,256)

    plotdata=[3.3*(float(x)/float(1024)) for x in ReadData]
```

```
# Set the screen background
screen.fill(BLACK)

scope_text = "CH1 = %s  M = %s " % (vpd_list[volt_Index], tpd_list[time_Index])

font = pygame.font.Font('freesansbold.ttf', 20)

scope_info = font.render(scope_text, True, PURPLE)

writePlace = scope_info.get_rect(center = (160, 220))

pygame.draw.circle(screen, RED, (0, 200), 5)

#columns
for i in range(0, 8):
    pygame.draw.line(screen, WHITE, [i*40, top], [i*40, bottom], 1)
#rows
for i in range(0, 6):
    pygame.draw.line(screen, WHITE, [left, i*40], [right, i*40], 1)

#=====plot scope data=====

for i in range(0, data_len):
    val = 240 - plotdata[i]/vpd_list_num[volt_Index]*40 - 40
    lines[i] = [x_axis[i], val]

#print lines

pygame.draw.lines(screen, GREEN, 0, lines, 2)

#=====end scope data=====

screen.blit(scope_info, writePlace)

# Limit to 100 frames per second
clock.tick(100)

# Go ahead and update the screen with what we've drawn.
pygame.display.flip()

# avoid the program hang upon exit
pygame.quit()
```