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

Vipin Venugopal (vv258), Ye Kuang (yk749)

Advisors: Dr. Bruce Land, Dr. Joseph Skovira

Degree Date: May 2019

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: PIC32 AND RASPBERRY PI INTERFACE

Authors: Vipin Venugopal (vv258), Ye Kuang (yk749)

Abstract: The goal of the project was to develop a cross platform system capable of performing real time tasks, while enabling 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 final application.

Executive Summary

The objective of this project was to develop a hybrid system which 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 major task involved in this was to build a library and communication protocol between the PIC32 and Raspberry Pi, which will enable the Raspberry Pi to access PIC32 peripherals. The developer can use these library calls to seamlessly integrate PIC32 peripherals with the peripherals of the Pi and simultaneously allow the Embedded OS to do system management.

The outcome of this project was the successful interfacing between the PIC32 and Pi, fully tested set of real-time functions, a PCB for PIC32 the fits the Raspberry Pi3 and a prototype of an object tracking robot. The library developed for interfacing between the PIC32 and Raspberry Pi, is generic enough to support a wide range of applications. This document covers all the design choices and decision made during the development of the system. A set of future improvements and suggestions for additional features have been included in the document. Since the project serves as a platform that can be used for building a wide range of applications, a huge emphasis was placed on documenting the features in terms of usability. ***Hardware Guide*** in Appendix contains the BOM, General Assembly and PCB Layouts for building additional boards. ***Software Guide*** in Appendix gives a detailed explanation of every function available in the library, including example usage, making it easy for anyone to jumpstart on any application using the library. The code consisting of the *PIC32Interface* python library, PIC32 firmware and module test cases are available at https://github.com/vv258/PIC-and-Pi-Interface. All relevant documentation including PCB files are also available in this repository.

Contributions

Sl no	Task	Vipin Venugopal	Ye Kuang
1	PIC32 peripheral selection and pin assignment *	✓	
2	Mode of Communication and protocol design *		✓
3	State machine design for Communication *	✓	✓
4	Firmware implementation on PIC for peripheral control	✓	
5	Python Library development for Raspberry Pi		✓
6	Communication Testing		✓
7	Module Testing of PIC32 firmware	✓	
8	PCB design for PIC32 board	✓	
9	Assembly and Testing of PIC32 board	✓	
10	Application development for Raspberry Pi		✓
11	Module Testing for Raspberry Pi Application		✓
12	Integrated testing for PIC and Pi Interface with custom board	✓	✓

**includes contributions by Zesun Yang (zy366)*

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 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 real-time environment while choosing from these two approaches.

The objective of this project was to develop a hybrid system which can provide optimum real-time performance at minimum development cost. The goal is to develop an end application to demonstrate these features and a human tracking robot was chosen as the target system.

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. 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 microcontroller 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 slave, and command response interface between the two.

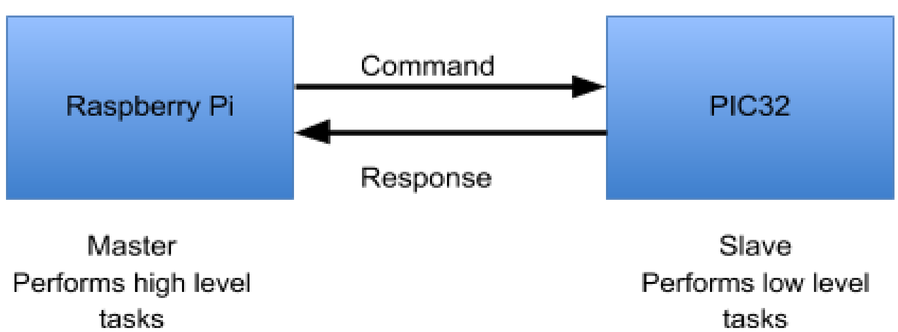


Figure 1. Proposed System

One application that can utilize the PIC32 and Raspberry Pi interface is a human tracking robot. Raspberry Pi's quad core CPU can be used to do computer vision and image processing, and then use vision information to track the owner of the robot, meanwhile sending control commands to PIC32 to control motors.

3 Design & Testing

The system development was done in various stages:

1) PIC32 peripheral selection and pin assignment
2) Mode of Communication and protocol design
3) State machine design for Communication
4) Firmware implementation on PIC for peripheral control
5) Python Library development for Raspberry Pi
6) Communication Testing
7) Module Testing of PIC32 firmware
8) PCB design for PIC32 board
9) Assembly and Testing of PIC32 board
10) Application development for Raspberry Pi
11) Module Testing for Raspberry Pi Application
12) Integrated testing for PIC and Pi Interface with custom board

3.1 PIC32 peripheral selection and pin assignment

Most of the PIC32's pins have multiple selectable functionalities, which needs to be configured based on the application requirement. In this project, the challenge is to identify the right combinations of these functionalities which could provide a range of features not available in the Raspberry Pi such as PWM motor control, sensor reading, and also a communication interface with Pi. Since many of the functionalities can be mapped only to a subset of the pins, choosing the right combination becomes critical due to the limited number of pins available on PIC32. The selection should not be limited to the specific application of smart suitcase, rather it should be generic to implement a wide range of applications.

Raspberry Pi has Bluetooth, Wi-Fi connectivity, data analysis and communication with PIC32 and USB interface but lacks in analog interfaces. Hence, the goal was to maximize the analog interfaces on the PIC32.

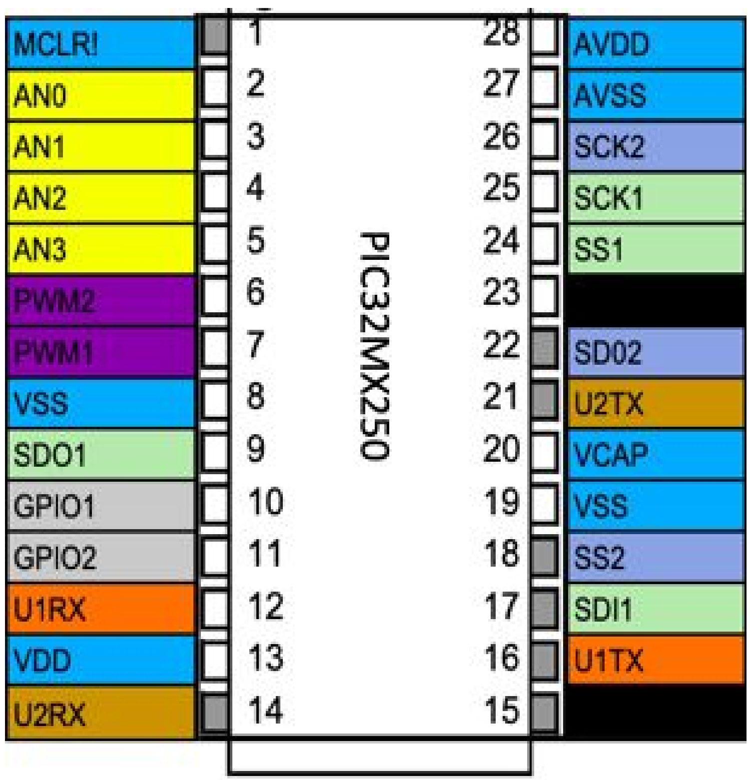


Figure 2. PIC32MX250 Pin Assignment

The pin assignment was first validated using microchip code configurator.

The Pins functionalities are explained in the table below:

Category	**Signal**	**Description**	**Pin No**
Supply and Reset	MCLR	Master Clear (Device Reset) Input.	1
	VSS	Ground Reference for Logic and I/O pins.	8,19
	VDD	Positive Supply for Peripheral Digital Logic and I/O pins.	13
	AVDD	Positive Supply for Analog Modules.	28
	AVSS	Ground Reference for Analog Modules.	27
	VCAP	External Filter Capacitor Connection	20
Analog Inputs	AN0	Analog Channel 0	2
	AN1	Analog Channel 1	3
	AN2	Analog Channel 2	4
	AN3	Analog Channel 3	5
PWM Outputs	PWM1	PWM Output 1	7
	PWM2	PWM Output 2	6
SPI	SDO1	SPI1 Serial Data Out	9
	SDI1	SPI1 Serial Data In	17
	SCK1	SPI1 Serial Clock	25
	SS1	SPI1 Slave Select	24
	SD02	SPI2 Serial Data Out	22
	SCK2	SPI2 Serial Clock	26
	SS2	SPI2 Slave Select	18
UART	U1RX	UART1 Receiver	12

	U1TX	UART1 Transmitter	16
	U2RX	UART2 Receiver	14
	U2TX	UART2 Transmitter	21
GPIO	GPIO1	General Purpose Input Output 1	10
	GPIO2	General Purpose Input Output 2	11

Table 1. PIC32MX250 Pin Description

3.2 Mode of Communication and Protocol design

Some of the options available for communication were UART, I2C, SPI and USB. Amongst all these options, USB is the fastest. However, the communication protocols for USB are complicated to design for the system. SPI is fast, but each SPI requires 4 pins and communication can only be initiated by the master. I2C requires fewer pins, but it is slow and the implementation can be complex. As a result, we chose UART as the mode of communication.

The first step in protocol design was to come up with the categories of commands. The initial categories of commands identified were System Commands, Expanded I/O commands, DAC Configuration and Control Commands, ADC Configuration and Control Commands, PWM Configuration and Generation Commands and Memory Commands.

The detailed description of the command categories are listed in the table below:

Slno	Category	Description
1	System	To initialize and maintain communication
2	Expanded I/O	To control the GPIOs in Port Expander connected to PIC
3	DAC	To configure and control the analog outputs in DAC IC connected to PIC
4	ADC	To configure the PICs internal ADCs and sample analog inputs
5	PWM	To configure the PICs internal Output Compare Unit to generate Pulse Width Modulated Signals
6	Memory	To write to and read from 4 1024 byte memory blocks inside PIC

Table 2. Command Categories

Once the broad categories were identified, the commands within each category were decided to provide a high level of configurability.

Sl no	Category	Command	Description
1	System	Handshake	Send a command and wait for response to check if communication is established
2	Expanded I/O	Read input	To read data from GPIO port Z of Port Expander
3		write output	To write data to GPIO port Y of Port Expander
4	DAC	Set Value for DAC Channel A	To Set DC value for DAC Channel A between 0 and 3.3V
5		Set Value for DAC Channel B	To Set DC value for DAC Channel B between 0 and 3.3V

6		ConfigDACA	To Configure DAC Channel A for arbitrary waveform generation
7		ConfigDACB	To Configure DAC Channel B for arbitrary waveform generation
8		ConfigDACA&B	To Configure DAC Channel A and B for arbitrary waveform generation
9		Start DAC	To start the generation of arbitrary waveform stored in any of the internal buffers from DAC
10		Stop DAC	To reset the DAC
11	ADC	Check Buffer Status	To check if ADC sampling is complete
12		Set Sample Frequency	To set ADC sampling frequency and number of samples to be acquired
13		Start ADC	To start the sampling of analog channels
14	PWM	Set Period	To Set Period for PWM
15		Generate PWM 1	To Set the ON time for PWM Channel 1 and generate signal
16		Generate PWM 2	To Set the ON time for PWM Channel 2 and generate signal
17	Memory	Read Buffer	To read data from Buffer memory in PIC32
18		Write Buffer	To write data to Buffer memory in PIC32

Table 3. Command Descriptions

The next task was to assign command codes to each command and specify the number of data bytes associated with each command. One important aspect of the protocol was to ensure that none of the data bytes get interpreted as the SOT and EOT. The SOT was chosen as F0 and EOT as D7. All data bytes had first 3 bits set to 0, so that F0 and D7 can never be sent as data. The response code for each command is same as the command code to check that the response is coherent with the command.

The detailed protocol is shown below:

Byte	**Value**	**Description**
SOT	F0	START OF TRANSMIT
EOT	D7	END OF TRANSMIT
CS	SUM OF COMMAND CODE+DATA BYTES	CHECKSUM

Table 4. Byte Description

Command	**RASPBERRY Pi to PIC**						
	BYTE1	BYTE2	BYTE3	BYTE4	BYTE5	BYTE6	BYTE7
Handshake	SOT	0x1A	EOT				
Read input	SOT	0x2A	CHECKSUM	EOT			
write output	SOT	0x2B	0000, 1 bit for each MSB I/O; 1- set	0000, 1 bit for each LSB I/O; 1- set	CS	EOT	

			HIGH 0- set LOW	HIGH 0- set LOW			
Set Value for CHA	SOT	0x3A	00,6 bit MSB value	00, 6bit LSB value	CS	EOT	
Set Value for CHB	SOT	0x3B	00,6 bit MSB value	00, 6bit LSB value	CS	EOT	
ConfigDACA	SOT	0x3C	000000,Buffer A number	000000, MODE	CS	EOT	
ConfigDACB	SOT	0x3D	000000,Buffer B number	000000, MODE	CS	EOT	
ConfigDACA &B	SOT	0x3E	0000Buffer A number,Buffer B number	000000, MODE	CS	EOT	
Start DAC	SOT	0x3F	00, 6 bits for prescaler	000, 5 bits for higher #of buffer samples	000, 5 bits for lower #of buffer samples	CS	EOT
Stop DAC	SOT	0X39	CS	EOT			
Check Buffer Status	SOT	0x4A	CS	EOT			
Set Sample Frequency	SOT	0x4B	00, 6 bits for prescaler	000, 5 bits for higher #of buffer samples	000, 5 bits for lower #of buffer samples	CS	EOT

Start ADC	SOT	0x4C	00, 4 bits: select analog channel, 2 bits:buffer selections	CS	EOT		
			6 bits	5 bits	5 bits		
Set Period	SOT	0x5A	000,time period	000,time period	CS	EOT	
Generate PWM 1	SOT	0x5B	000,on time	000,on time	CS	EOT	
Generate PWM 2	SOT	0x5C	000,on time	000,on time	CS	EOT	
Read Buffer	SOT	0x6A	000000,Buffer number	000, 5 bits for higher #of buffer samples	000, 5 bits for lower #of buffer samples	CS	EOT
Write Buffer	SOT	0x6B	000000,Buffer number	000, 5 bits for higher #of buffer samples	000, 5 bits for lower #of buffer samples	CS	EOT

Table 5. Raspberry Pi to PIC protocol

RESPONSE	**PIC to RASPBERRY Pi**				
	BYTE1	BYTE2	BYTE3	BYTE4	
Handshake	SOT	0x1B	EOT		
Read input	SOT	0x2A	0000,1 bit for each MSB I/O; 1- high, 0- low	0000, 1 bit for each LSB I/O; 1- high, 0- low	
write output	SOT	0x2B	EOT		
Set Value for CHA	SOT	0x3A	EOT		
Set Value for CHB	SOT	0x3B	EOT		
ConfigDACA					
ConfigDACB					
ConfigDACA&B					
Start DAC					
Stop DAC					
Check Buffer Status	SOT	0x4A	0000, 1bit for each of four buffers, 1- ready, 0 -not ready	EOT	

Set Sample Frequency	SOT	0x4B	EOT		
Start ADC	SOT	0x4C	EOT		
Set Period	SOT	0x5A	EOT		
Generate PWM 1	SOT	0x5B	EOT		
Generate PWM 2	SOT	0x5C	EOT		
Read Buffer	SOT	0x6A	EOT	followed by DMA burst	
Write Buffer	SOT	0x6B	EOT	followed by DMA burst	0x3A as Byte ACK for every 8th received Byte

Table 6. PIC to Raspberry Pi protocol

3.3 State machine design for Communication

The State Machine works on a command-acknowledgement basis. The protocol has the provision to be robust (error free) using status and control features such as SOT, EOT, invalid commands and checksum to indicate communication status. Receiving data by PIC can only be done using a state machine as the protocol uses commands of different length and the PIC does not know the length of command until the command code is parsed. A state machine implementation was not required for the Raspberry Pi as the Pi is the master and initialises all communications. The command and associated number of data bytes are known to the Pi before transmission. Since command response occurs in-order, the length of response is also known to Pi.

The states in PIC receiver are the following:

State	Description
ACTIVE_WAIT_FOR_SOT	Waits for receiving SOT
ACTIVE_PARSE	Reads the command code. Calculates the number of data bytes associated with the command code. Reads the data bytes
ACTIVE_CHECKSUM	Calculates checksum by adding command code and data bytes. Compares with the received Checksum
ACTIVE_WAIT_FOR_EOT	Waits for receiving EOT
EXECUTE	Executes the command and sends the response message. For read buffer alone, the response is sent first and then the command is executed.

Table 7. State Description

The state diagram is shown below:

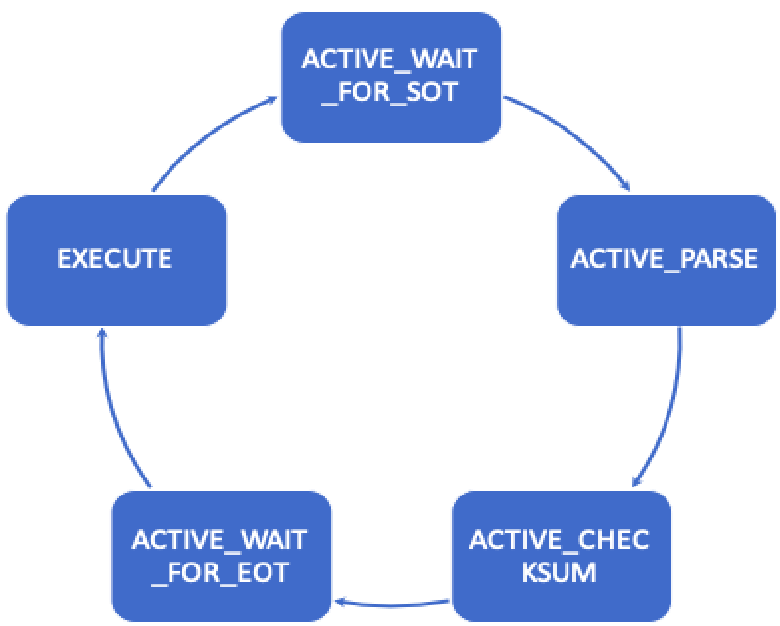


Figure 3. State Diagram

3.4 Firmware implementation on PIC for peripheral control

The various components of firmware for PIC32 consists of the following

a. Pin Assignment PPS Input/Output
b. Setup of Peripherals like SPI, UART, TIMERS, PWM, ADC
c. UART channel for continuous debug and troubleshooting
d. State Machine for communication
e. Functions for controlling peripherals

Peripheral Input Select is used to configure the following inputs and outputs:

Type	Group	Signal	Pin
Input	3	U1RX	RPA4
Input	2	U2RX	RPB5
Input	2	SDI1	RPB8
Output	1	U1TX	RPB7
Output	1	OC1	RPB3
Output	4	SS2	RPB9
Output	4	U2TX	RPB10
Output	3	OC4	RPB2
Output	2	SDO2	RPB11
Output	3	SDO1	RPA2

Table 8. PPS Configuration

The peripherals were configured with the following settings:

UART1 is used for displaying debug message. It is configured for using only TX and RX pins, with 8 bit data, no parity, 1 stop bit and 9600 baud rate. UART2 is used for communication with Raspberry Pi. It is configured for using only TX and RX pins, with 8 bit data, no parity, 1 stop bit and 115200 baud rate.

ADC module is configured with output in integer format, auto triggering and auto sampling. External reference voltage is used for ADC with offset test disabled, and scan mode enabled. ADC samples 4 times per interrupt with dual buffers and using only MUXA, internal reference clock and sample times as 15. Channels AN0,AN1, AN2 and AN3 are configured as analog inputs and all other channels are added to skip scan list.

SPI1 is used to interface with the Port Expander IC. It is configured with prescalar 4, 8 bit data mode, PIC as master and reverse polarity between data and clock. SPI2 is used to interface with the DAC IC. It is configured in framed mode with prescalar 4, 16 bit data mode, PIC as master and reverse polarity between data and clock.

DMA Channel 1 is set up to transfer into the receive buffer every time an interrupt occurs at UART2 receiver. DMA Interrupt is triggered when a pattern match with EOT occurs. The communication state machine runs inside the DMA interrupt. This allows the UART data to be received using only hardware and allows the software to respond to it whenever it is free. Otherwise, this would lead to data loss.

When the command for setting ADC sampling is received, TIMER1 is setup with the sampling frequency and timer interrupt is enabled. The mapping of Analog channel to PIC buffer is done with the start ADC command. The sampled values are copied to the buffer inside the interrupt. When the desired number of samples are received the interrupt is disbaled and the buffer status is set. The Pi can check completion of sampling by reading the buffer status.

When the command for setting PWM period is received, TIMER2 is setup with Period as the PWM period. The prescalar value is scaled appropriately to allow a dynamic range. The output Compare Units 1 and 4 are setup with TIMER2 as source and ON time as 0. When Start PWM command is received, the ON time of corresponding OC module is set to the desired ON time.

For reading and writing the GPIOs in Port expander, SPI transactions are used. The SPI read operation is started by lowering CS. The SPI read command (slave address with R/W bit set) is then clocked into the device. The opcode is followed by an address, with at least one data byte being clocked out of the device. The SPI write operation is started by lowering CS. The Write command (slave address with R/W bit cleared) is then clocked into the device. The opcode is followed by an address and at least one data byte.

For setting DAC voltages, 2 types of operations are used, depending on whether the DAC is used to set fixed dc value or to generate arbitrary waveforms. For fixed dc values, the write transaction is used in normal spi mode. The write command is initiated by driving the CS pin low, followed by clocking the four Configuration bits and the 12 data bits into the SDI pin on the rising edge of SCK. The CS pin is then raised, causing the data to be latched into the selected DAC's input registers. The configuration bits are changed depending on the channel to be used. For arbitrary waveform generation, first each sample data is appended to the configuration bits and stored in a buffer. If both channels are simultaneously used, then the samples are stored in alternate locations of the buffer. TIMER3 is set up with frequency same as the synthesis frequency for single channels and double synthesis frequency for dual channel modes. DMA channel is setup to transfer the data to the DAC over SPI in frame mode. The DMA channel is opened in default mode or auto mode depending on whether the generation mode is single burst or continuous. When a Stop DAC command is received, the DMA channel is disabled and TIMER3 is closed.

To display debug messages, the message is sent through the UART channel one character at a time and a newline character is sent at the end of the message. After executing the command, the response message is sent by sending the SOT, followed by command code, data bytes if any and then the EOT. For Read Buffer alone, the response is sent before executing the command, to help the Pi to distinguish between acknowledgement and response data.

3.5 Python Library development for Raspberry Pi

The python library was developed in accordance with the communication protocol. The library was named as ***PIC32Interface***. It consists of:

1. Setup of serial port
2. Helper functions for Checksum & Data transfer/reception
3. Functions for controlling PIC peripherals

The serial port is setup using Python Serial module with following settings:

Channel: /dev/ttyAMA0

Baudrate: 115200

Parity: None

Stopbits: 1

Byte size: 8 bits

For testing from PC, the serial port channel may have to be changed accordingly.

The two helper functions available in the library are Chceksum and SendCommand.

Checksum calculates sum of command code and data bytes to be sent along with the command. The SendCommand function takes the command code, data bytes and checksum, adds SOT to the beginning and EOT to the end and converts the bytes to hex array. The hex bytes are then written to the serial port.

The following functions are available for the user to control PIC peripherals from Pi:

1. WriteBuffer
2. ReadBuffer
3. ReadBuffer2
4. SetPWMPeriod
5. EnablePWM1
6. EnablePWM2
7. SetDACA
8. SetDACB
9. ConfigureDACA
10. ConfigureDACB
11. ConfigureDACAB
12. StartDACOutput
13. StopDACOutput
14. CheckBufferStatus
15. _StartADC_
16. _SetSampleFreq_
17. WriteGPIO
18. ReadGPIO

The detailed instructions for using these functions in application is included in the Software User guide in Appendix.

3.6 Communication Testing

Since the mode of communication is UART, the communication testing could be carried out independently for both PIC and Pi. Docklight software was used along with USB to serial converters to carry out module testing. For PIC testing commands were sent from Docklight, the interpreted message was displayed on the debug UART channel using another docklight window. For Testing the Raspberry Pi, the sent message from Pi was printed on its terminal and received a message on a Docklight window. These two were then compared for verification.

The communication was happening at 115200 baud rate for most commands. However, for PIC32 for large data reception, this resulted in framing errors. Initially, the baud rate for communication was set at 9600 to overcome this issue. Towards the end, this was resolved by adding an additional acknowledgement for every 8th byte received, baud rate of 115200 was achieved.

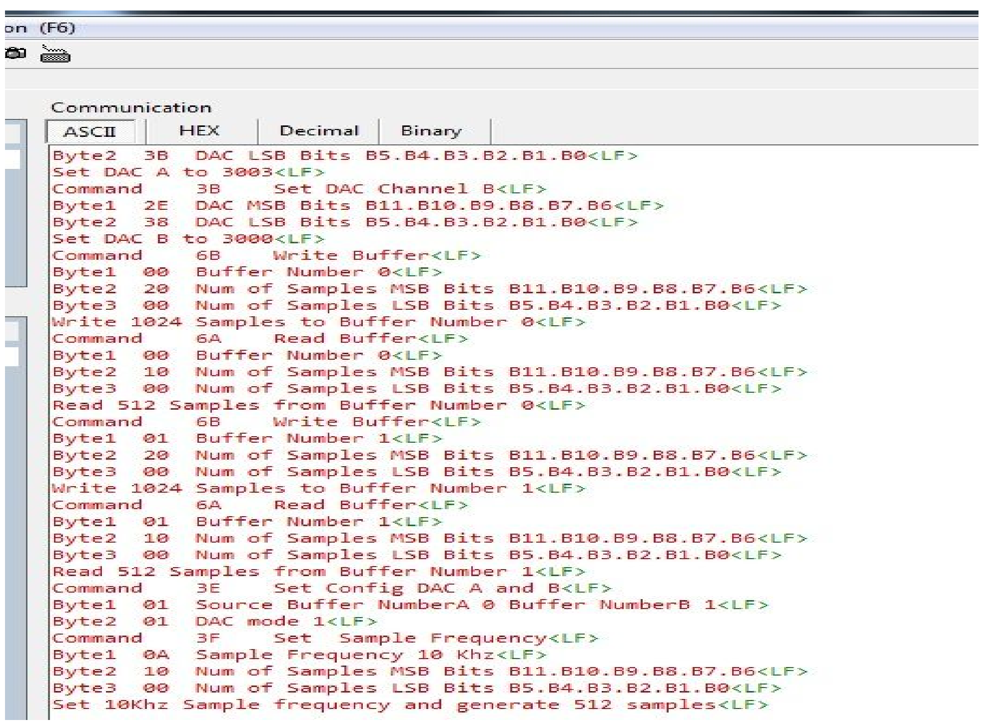


Figure 4. Debug window showing PIC-PI communication and data transfer

3.7 Module Testing of PIC32 firmware

The choice of UART for communication proved quite useful module testing. PIC functionality could be tested independent of Pi by using a PC/USB-Serial Converter. Test cases were written for testing the following. The test cases were integrated into a Jupyter Notebook for ease of testing. It enabled quick editing of test parameter and running of code blocks for the following cases:

1) Data transfer
2) PWM Generation
3) DAC Waveform Generation
4) ADC sampling
5) GPIO control

The Memory Read/Write was tested by writing a chunk of data into PIC memory and reading it back over serial communication. These were then compared for verification.

PWM generation was first tested by viewing the generated PWM signal on oscilloscope. It was further testing by driving continuous rotation Parallax servo motors are various speed. One issue observed was that PWM was not getting generated for all ON time values. It was due to the limited range of base TIMER setup. This issue was resolved by adding a dynamic prescaling for the base TIMER with respect to the PWM ON time & Period.

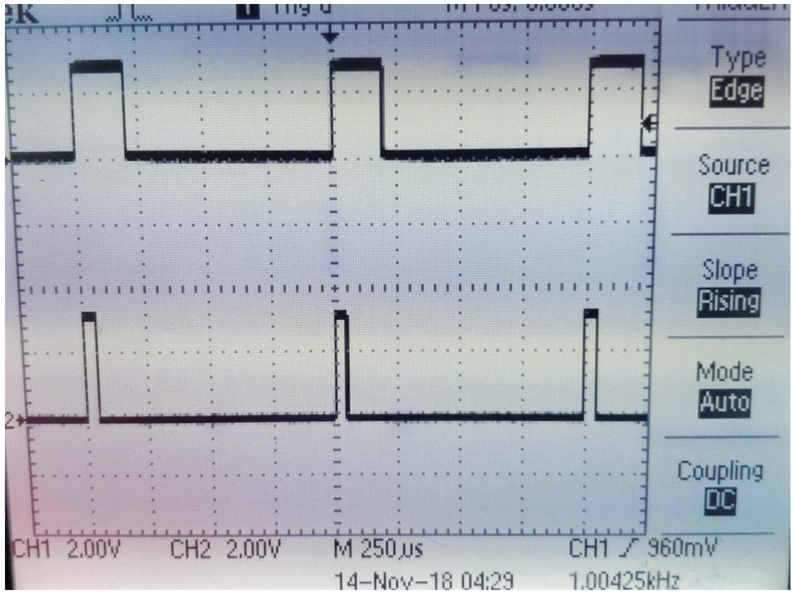


Figure 5. PWM generation

The DAC was tested for generating a fixed DC value, as well as arbitrary waveforms like sine and triangular waveform. These arbitrary waveforms were first written into the PIC buffer and then played out in both burst mode and loopback mode.

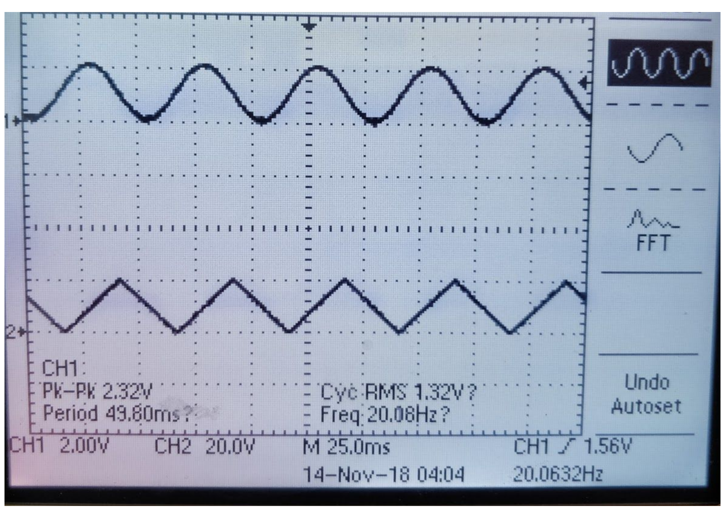


Figure 6. DAC generation

GPIO testing was carried out using loopback. The GPIO input channels and output channels were treated as parallel ports to send and receive data. Data matching was carried out to verify this.

ADC sampling was tested using both fixed DC input values and waveforms like sine, square and triangular waveforms for all 4 channels. The plot of sampled AC waveform was initially highly distorted. This was due to the printing of debug message in the ADC interrupt leading to delays. The next interrupt was getting generated before completion of printing. This was resolved by removing the debug message in interrupt.

3.8 PCB design for PIC32 board

A protoboard mating with the PIC32 small board was wired up to ensure that the connections are correct before doing the final board design. Pin connections were verified using this setup by carrying out all module tests.

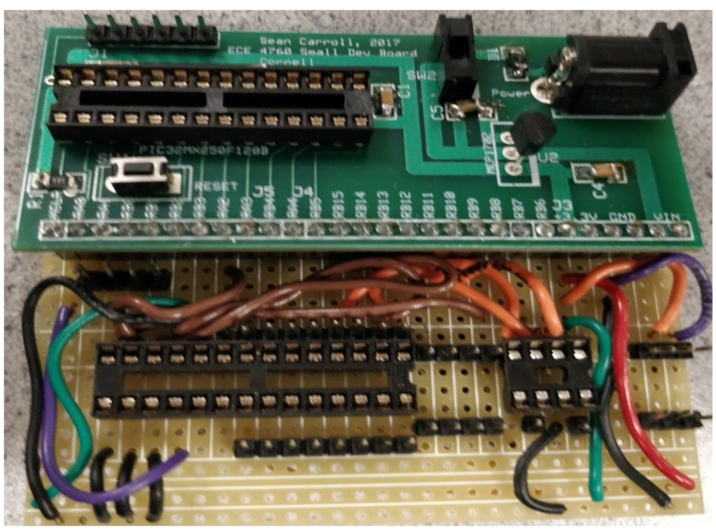


Figure 7. Protoboard

The final schematic was designed using ExpressSchematic and ExpressPCB was used for layout. The schematic and layout are available in the Appendix. The following aspects were taken into account for the design.

a. The overall dimension of board should be same as RaspberryPi
b. It should mate with the 40 pin header of Pi. At the same time, these pins have to accessible for connecting Pi display. This was done using stacking connectors.
c. Use of through hole components to ensure Design for Assembly and Design for Maintainability.
d. Provision to power up PIC from external source or from Pi. This was implemented using optional resistor.
e. Optional connections for PIC programming pins to RPi GPIOs for future scope.
f. Pin arrangement for ease of connection with parallax servo motors and USB to serial converters.
g. External supply input for driving motors.
h. Sufficient number of supply and ground pins for connecting external sensors and actuators and reducing the wiring required.

3.9 Assembly and Testing of PIC32 board

The board was assembled and tested for full functionality. The only issue in the board was related to the Power ON LED, which had both ends connected to ground. However, this was not a major issue as it did not affect functionality.

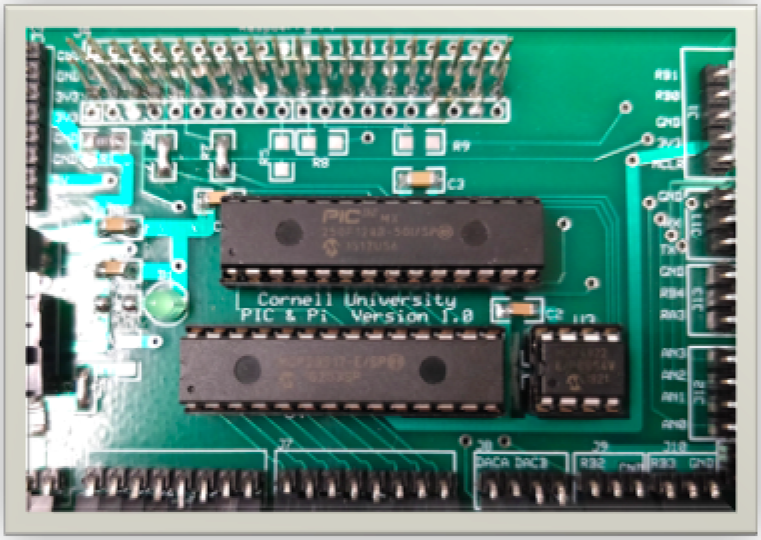


Figure 8. Final board

Detailed Bill of Materials and General assembly details are given in the Hardware Guide included in Appendix. The PCB and schematic files are shown in Appendix and also included in the repository.

The board takes less than 30 minutes to assemble. Testing was carried out using Jupyter Notebook available in repository.

The power output points on the board was verified using multimeter.

UART communication and debug was verified by observing the debug message display.

PWM generation was first tested by viewing the generated PWM signal on oscilloscope. It was further testing by driving continuous rotation Parallax servo motors are various speed.

The DAC was tested for generating a fixed DC value, as well as arbitrary waveforms like sine and triangular waveform. These arbitrary waveforms were first written into the PIC buffer and then played out in both burst mode and loopback mode. This also validates Memory Write.

ADC sampling was tested using both fixed DC input values and waveforms like sine, square and triangular waveforms for all 4 channels. This also validates memory read.
GPIO was tested using loopback check.

For testing a newly assembled board, a Test Plan is provided in the Appendix.

3.10 Application development for Raspberry Pi

The application for the cross platform system is a human tracking robot, it features human tracking based on computer vision(QR code tracking). Raspberry Pi will do all the image processing and decision making, while PIC32 generates PWM to control motors. The hardware components needed are listed below:

1) PIC32 board
2) Raspberry Pi 3 model B
3) Two parallax standard servos
4) Robot frame
5) Pi camera

The initial development was carried out with Pi's built-in PWM.

3.10.1 QR code tracking

To track the owner of the robot, we need some unique target pattern to help the robot recognize the owner. At first, we chose to use circles with different colors, but it turned out to be too slow that we can only get 2 or 3 frames of processed images per second, it will cause trouble for our control loop since the delay is high, not to mention color tracking is greatly influenced by environment lighting . So we came up with an alternative solution, which is

QR code tracking. QR code tracking is less computational intense than shape and color tracking, and it's also pretty accurate.

Figure 9. QR code recognition

Figure 9 shows how our QR code tracking works, the program captures the specific QR code in the frame and calculate its center point for further use.

3.10.2 Multiprocessing

To accelerate image processing and make full use of Raspberry Pi's quad core CPU, we took the strategy of multiprocessing(inspired by Autonomous Turret Tracking Project, see appendix), which enables us to process 3 frames of images simultaneously, the structure of our multiprocessing strategy is as below:

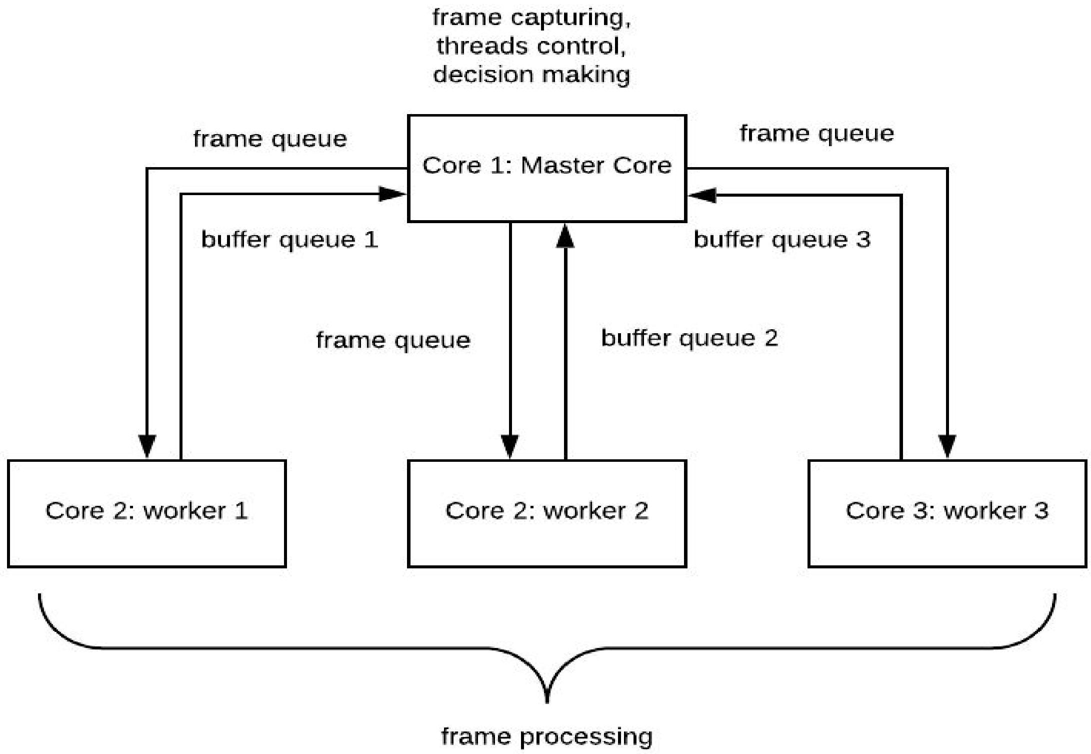


Figure 10. Multiprocessing structure

As you can see in Figure 10, three processors are assigned as worker processors and one is master processor. Worker processors' only job is processing frames captured by Pi camera and send the result back to master processor. Master processor is in charge of frame capturing, coordination of worker processors and decision making based on information collected from workers.

Processes communication is handled by queue, one frame queue is used for master processor to handle frames to each worker processor, and three buffer queues for worker processors to send back frame processing result to master processor. Process lock technique is applied to ensure correct workflow.

3.10.3 Motor control

The motor we are using is parallax standard servo. The Parallax Standard Servo is controlled through pulse width modulation, where the position of the servo shaft is dependent on the duration of the pulse. In order to hold its position, the servo needs to receive a pulse every 20 ms. Figure 11 is a sample timing diagram for the center position of the Parallax Standard Servo.

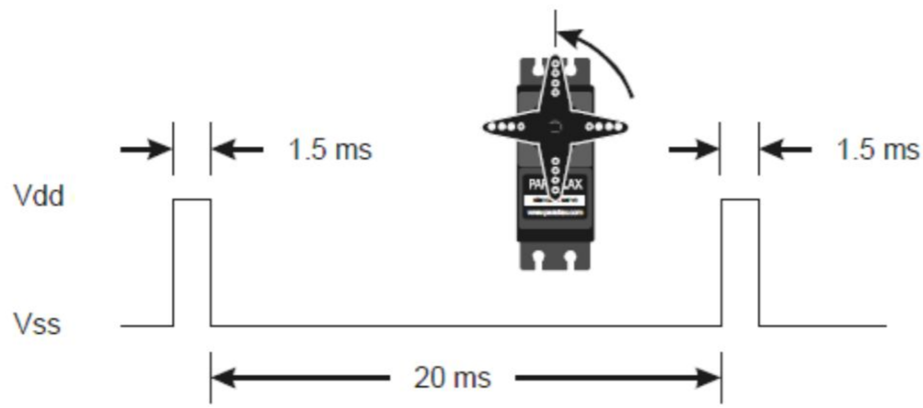
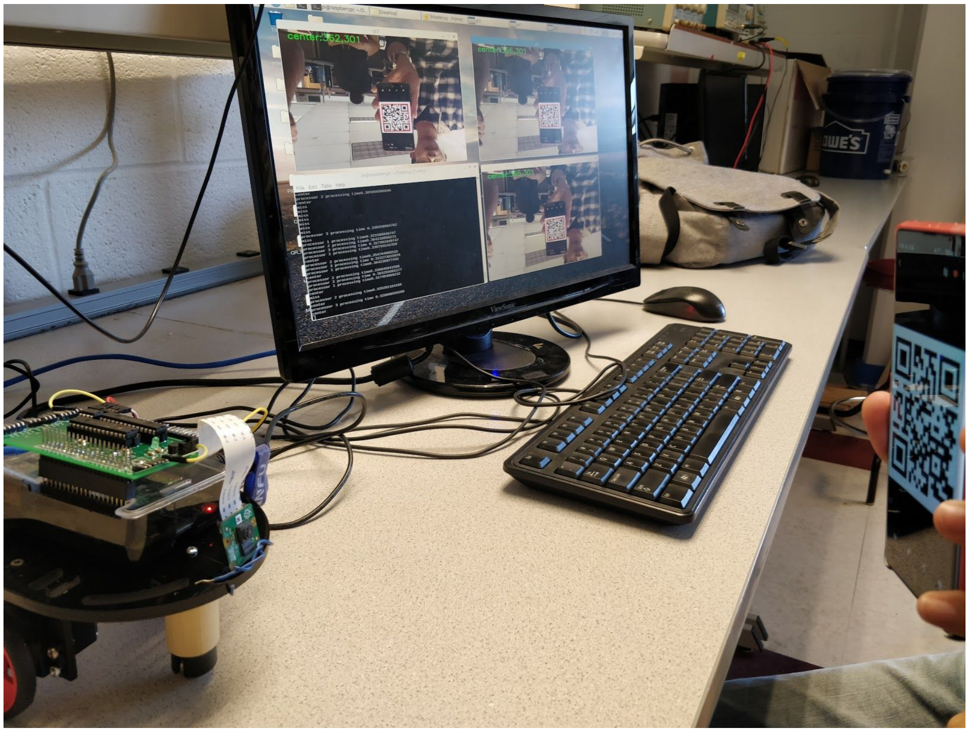


Figure 11. parallax standard servo timing diagram from data sheet

3.11 Module Testing for Raspberry Pi Application

The first stage of testing was for the camera interface. Initially, OpenCV was used to detect multiple circles of different colors. However, the processing time for circles were quite high. This meant that we would not be able to achieve real-time performance. By switching to QR code scanner, we were able to achieve 200ms processing time (5 frames per second).

Next the motors were calibrated. Extensive tuning was carried out to set the speeds for servo motors. High speed would lead to overshooting and the QR code going out of frame. Slow speed meant that the system responsiveness is quite low and would not be able to track fast movements of target.



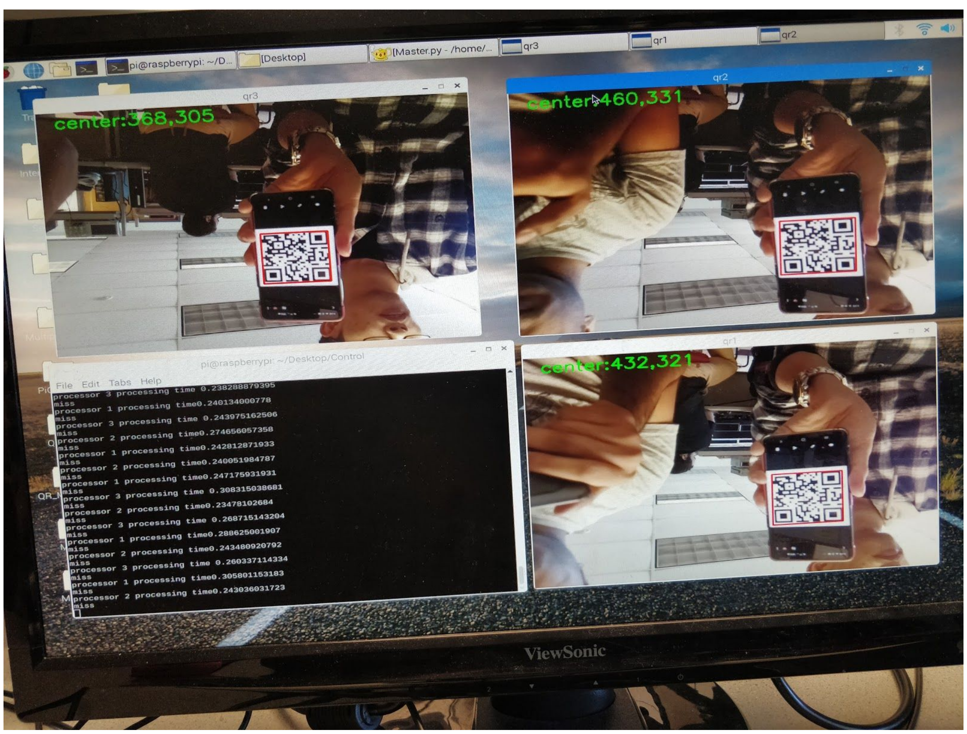


Figure 12. Simultaneously tracking 3 frames

3.12 Integrated testing for PIC and Pi Interface with custom board

The integration was comparatively easier as the modules and application were independently tested prior to this. The Pi PWM is configured using frequency and duty cycle, whereas, the ***PIC32Interface*** PWM is configured using Period and ON time. The only change in the code was to do this mapping. The performance of the system was comparable to the one using built-in PWM modules.

4 Results

During the past two semesters, we spent a lot of effort in exploring project development as well as team management. At the end of the second semester, we have achieved some substantial milestones, which we are enthusiastic to summarize in this section.

The PIC32 peripherals were selected and mapped to the pins(see Figure 13). The mapping was validated using Harmony Code configurator. There is the very first thing we did in the development of our project, since almost everything else depends on the appropriate pin mapping.

A list of real-time functions were selected and a robust protocol was designed to use the UART mode of communication. We drafted the communication protocol together and refined it with the help of Professor.Land. We also designed the corresponding state machine for implementing the protocol for both PIC and Pi.

After the protocol and state machine design, we implemented the protocol on both sides using python and also validated the protocol using serial helper tool. The peripheral control functions of PIC were implemented in C and verified too.

After protocol code is implemented, Vipin designed a PIC32 PCB (see Figure 14) that mates with Raspberry Pi was designed, manufactured, assembled and tested successfully. The PCB was mounted on Raspberry Pi (Figure 16) and tested for all functionality.

An end application of a QR code tracking robot was designed and tested on Raspberry Pi. We tried many ways of tracking algorithm, at last we chose QR code tracking for our application since it has the best performance among all methods we tried. Pi camera and OpenCv library is used for QR code tracking. The PIC and PI were then integrated and installed on the robotic platform and final application using integrated system was successfully tested (see Figure 17).

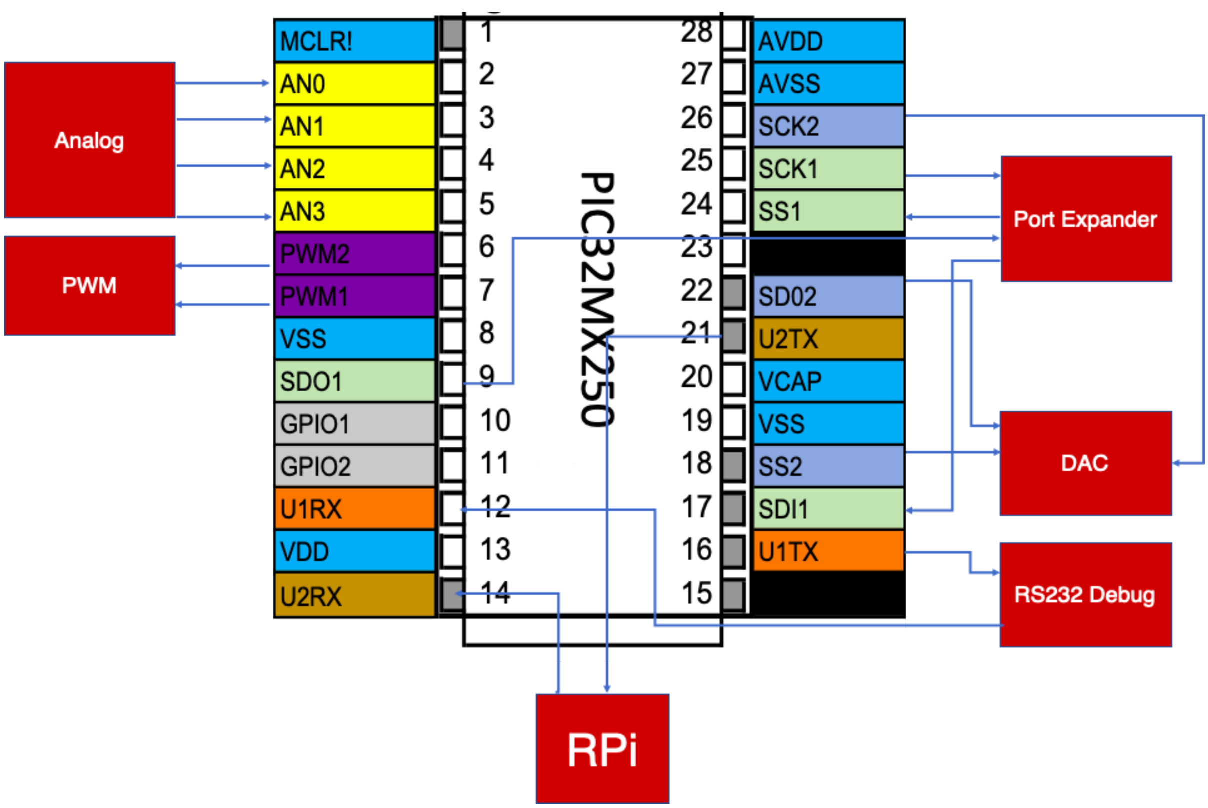


Figure 13. overall system

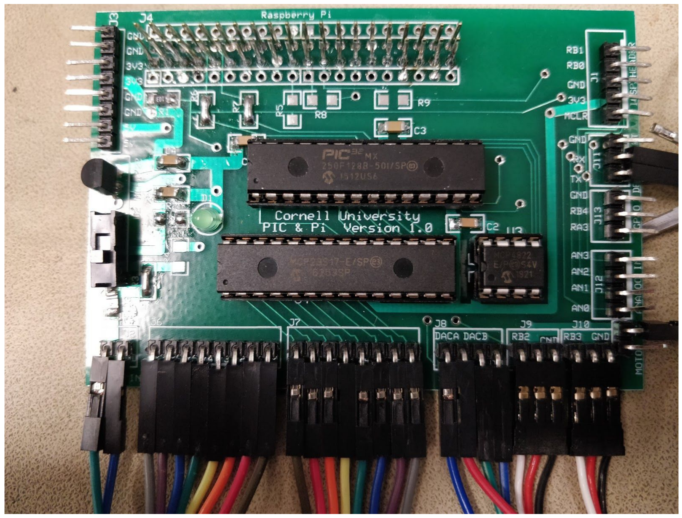


Figure 14. PIC32 board

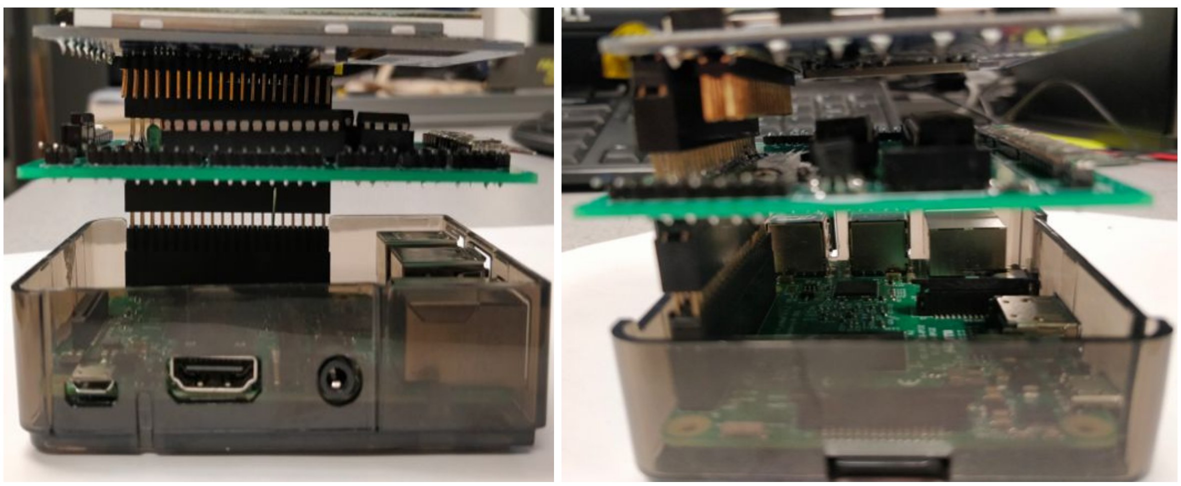


Figure 15. PIC32 and Raspberry Pi stack side views

Figure 16. PIC32 and Raspberry Pi stack

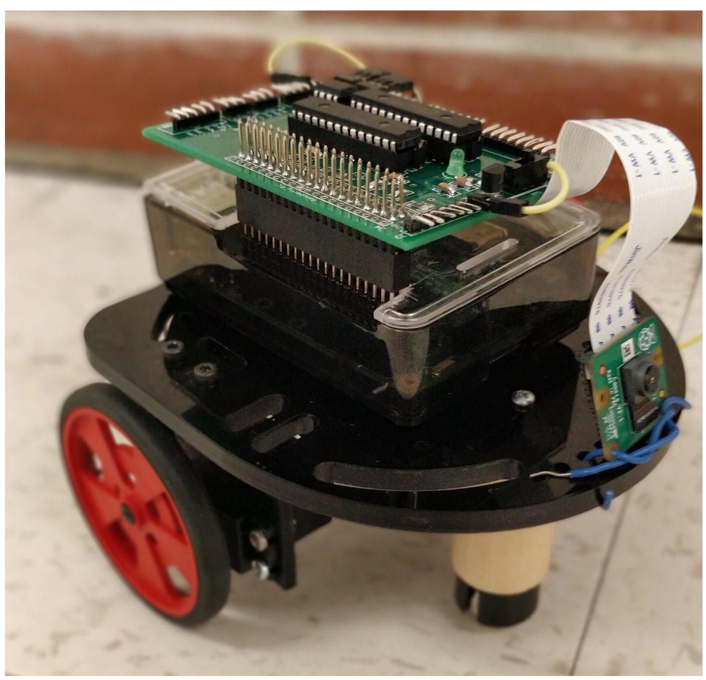


Figure 17. Integrated Robot

5 Future improvements

5.1 Protocol/ State Machine and functions

The first thing we need to add into our communication protocol implementation is checksum. A checksum is a string of numbers and letters that act as a fingerprint for a file against which later comparisons can be made to detect errors in the data. They are important because we use them to check files for integrity. The protocol has framework for checksum. But this is not implemented. Implementing this can make the protocol robust.

Other desired improvements are all trivial but we haven't got time to implement them yet. Right now The Pi receives the acknowledgement. However, currently it is only read. There is no validation for this, it's important to add a validation process for acknowledgement. Pin number 10 and 11 of PIC are currently unused. Additional functions can be implemented to utilise this. The python interface library we wrote should be encapsulated into a class, the objects can be initialized by specifying the serial ports. Using classes will make our code more organized and turn it into generic reusable pieces. This would enable us to connect multiple PICs to RPi using the USB ports of Pi. Also we need to write equivalent C library which can improve code performance and increase utility.

5.2 PIC32 PCB

For PIC32 PCB, we also have a few improvements in mind. In the present version of the PCB, both ends of Power ON LED are connected to ground. This has to be corrected in the future versions.The PCB can be redesigned to using SMD components. A USB to serial convertor can be integrated into the PCB. This can then be used as a dongle to add analog and other interfaces to any Desktop/Laptop/Embedded Platform. Adding a heat sink to Raspberry Pi CPU is also important, right now CPU is easy to overheat and throttle, which results in performance degradation, if we want to keep a constant performance, cooling the component is necessary

5.3 Remote Programming

Currently there is a hardware provision to connect PIC programming pins to RPi GPIOs. With proper scripting in PI, this interface can be used to remotely program the PIC using Pi. On running update PIC command on Pi, the script should be able to download the latest code from git, compile using GCC and program PIC using GPIO.

5.4 End Application

Our application is a prototype at the moment and it needs tuning and refinement. The first thing that needs to be done is adding a feed-back loop to motor control, right now it's a open-loop control system, feed-back loop control will greatly improve tracking accuracy. And we want to make use of distance sensor(e.g.,ultrasonic sensor) to get distance information, and add this distance information to feed-back loop to keep a constant distance between robot and human, which will also increase tracking accuracy.

Although QR code tracking is great and it performs relatively well on our system, we want something better. Professor.Land mentioned to us there is something called AprilTag, it is designed for robot tracking system and it's less computational intensive than QR code tracking, if we replace QR code tracking with Apriltag tracking, the performance of our system will definitely increase.

And the camera we are using right now is the normal Pi camera, its field of view is very narrow, we want to replace it with fisheye camera which has over 100 degrees of field of view, but fisheye camera needs calibration to recognize patterns, since it will deform patterns. We need to write some program and adapt checkerboard pattern calibration technique to calibrate fisheye camera. And IR beacons can be integrated into the system to level coarse direction information when the tag is out of field of view.

5.5 Other Applications

Current end application utilises only the PWM. Other real-time applications like oscilloscope or inverted pendulum which can use full capability of the interface can be implemented.

6 Conclusion

The project is a success. The PIC32 and Raspberry Pi interface is versatile and easy to use, which greatly increases the potential of our cross platform system, one can easily create many electronic devices such as an oscilloscope using our system. And our human tracking robot application based on this system serves as a proof of concept, it works great and shows us how useful the combination of Raspberry Pi and PIC32 is. There are many foreseeable improvements to be carried out. We believe this system will grow better, and be useful to everyone who is interested in embedded devices development.

7 Acknowledgements

We would like to express our deepest appreciation to all those who helped with this project. A special gratitude we give to our project advisors, Professor. Bruce Land and Professor. Joseph Skovira, whose contribution in stimulating suggestions and encouragement, helped me to coordinate my project especially in writing this report. We would also like to thank Zesun Yang for her contributions to the initial phase of the project, and Autonomous Turret Tracking project inspired us to use multiprocessing to accelerate image processing, many thanks for him sharing his code. Several portions of the PIC firmware are modifications of code snippets spread across the ECE4760 course website.

8 References

1.ECE 4760 Course Website

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

2. PIC32 reference board

http://people.ece.cornell.edu/land/courses/ece4760/PIC32/target_board.html

3. PIC32 datasheet

http://people.ece.cornell.edu/land/courses/ece4760/PIC32/Microchip_stuff/2xx_datasheet.pdf

4. PIC32 Reference manual

http://ww1.microchip.com/downloads/en/devicedoc/61113e.pdf

5. PIC32 Peripheral Library Guide

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

6. Port Expander datasheet

http://people.ece.cornell.edu/land/courses/ece4760/PIC32/Microchip_stuff/port_expander.pdf

7. DAC datasheet

http://ww1.microchip.com/downloads/en/DeviceDoc/20002249B.pdf

8. ECE5725 Course Website

http://skovira.ece.cornell.edu/ece5725/

9. PiCamera Document

https://picamera.readthedocs.io/en/release-1.13/

10.RaspberryPi

https://www.raspberrypi.org/

11. Parallax Continuous Rotation Servo

https://www.parallax.com/sites/default/files/downloads/900-00008-Continuous-Rotation-Servo-Documentation-v2.2.pdf

12. Autonomous Object Tracking Turret

https://courses.ece.cornell.edu/ece5990/ECE5725_Spring2018_Projects/fy57_xz522_AutoTurret/index.html

13.Pyzbar tutorial

https://www.learnopencv.com/tag/pyzbar/

14.Raspberry Pi uart setting

https://www.raspberrypi.org/documentation/configuration/uart.md

Appendix A. Hardware Guide

All relevant documentation including PCB files are also available at https://github.com/vv258/PIC-and-Pi-Interface.

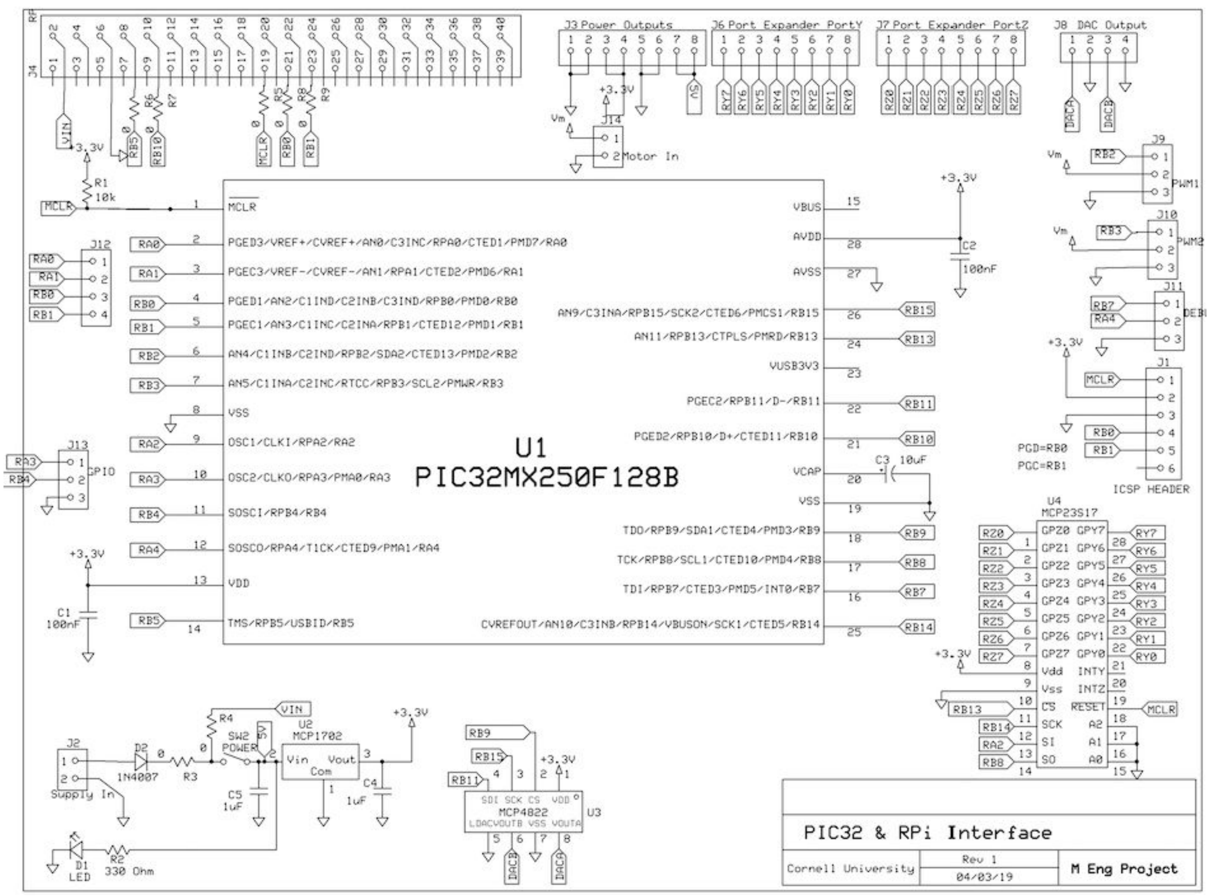


Figure A-1. PIC board schematics

Sl no	Reference	Description	Remarks
1	C1, C2	100nF	
2	C3	10uF	
3	C4, C5	1uF	
4	D1	LED	
5	D2	1N4007	
6	J1-J3,J6-J14	Right angle male headers	ICSP HEADER
7	R1	10k	
8	R2	330 Ohm	
9	R3(#)	0	Mount for powering PIC from external supply
10	R4(# $)	0	Mount for powering PIC from RPi
11	R6, R7	0	
12	R5,R8,R9(*)	0	Mount for programming from Pi
13	SW2	POWER	
14	U1	PIC32MX250F128B	
15	U2	MCP1702	
16	U3	MCP4822	
17	U4	MCP23S17	
18	J4	40 pin stacking connector	
	# Do not mount R3 and R4 at the same time		
	$ Not tested yet. RPi may not be able to supply sufficient power		
	** Not tested yet*		

Table A-1. PIC board Bill of Materials

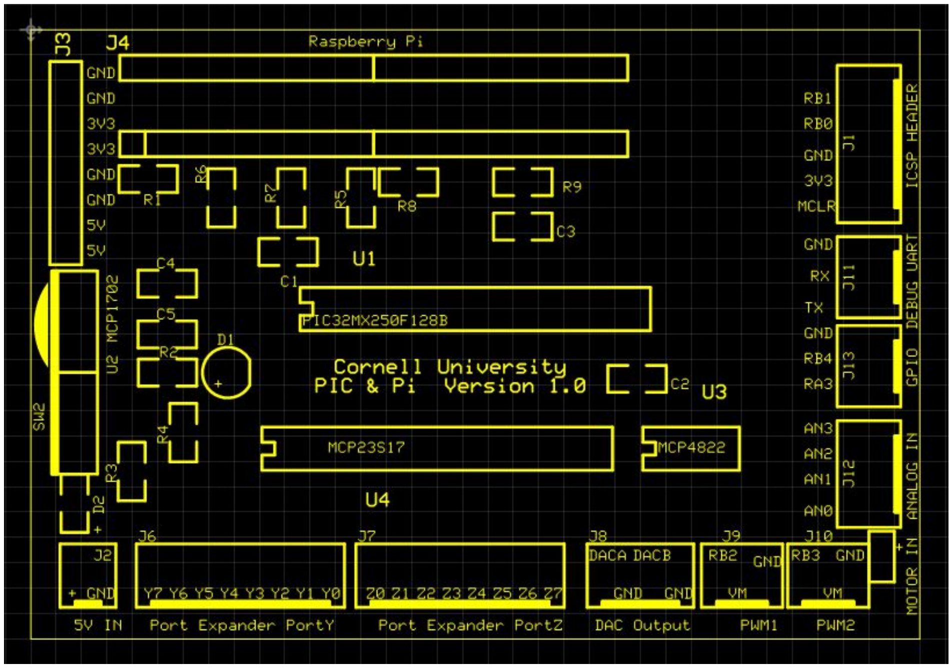


Figure A-2. PIC board General Assembly

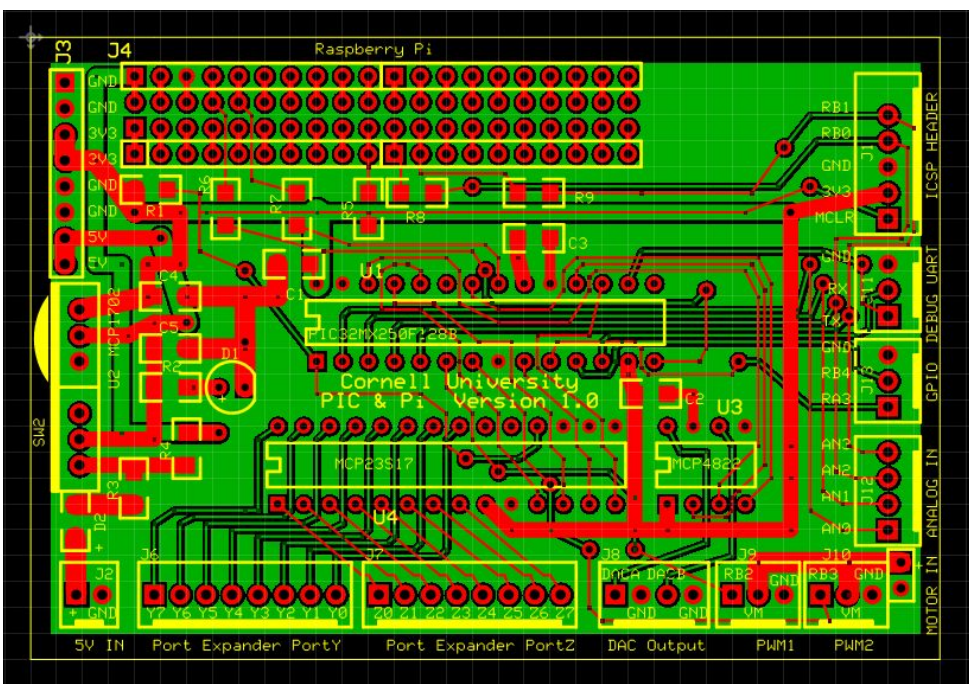


Figure A-3. PIC board PCB Layout

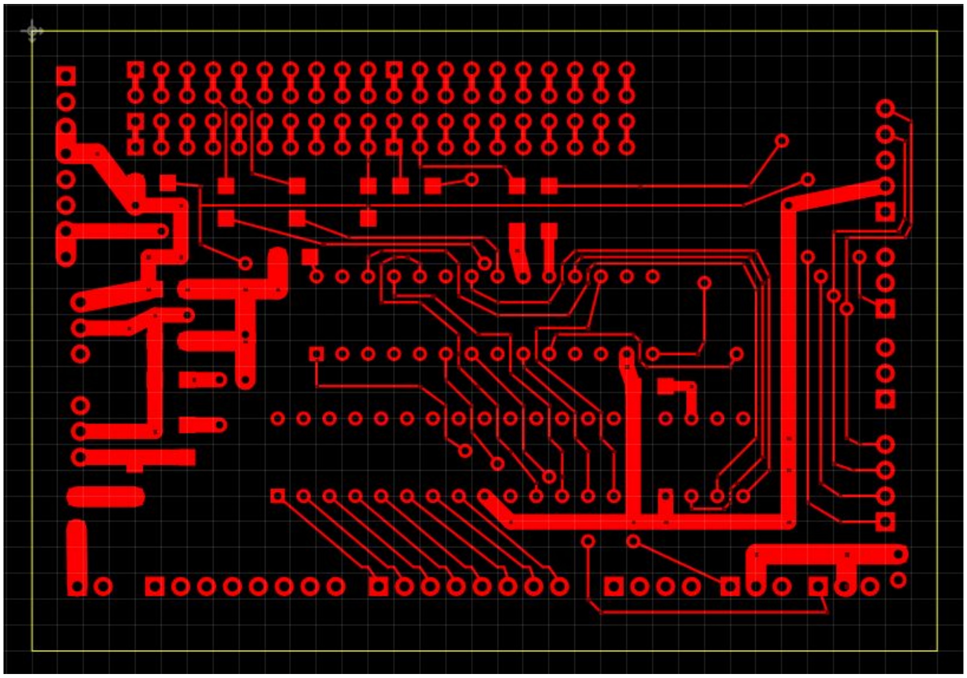


Figure A-4. PIC board Top Layer

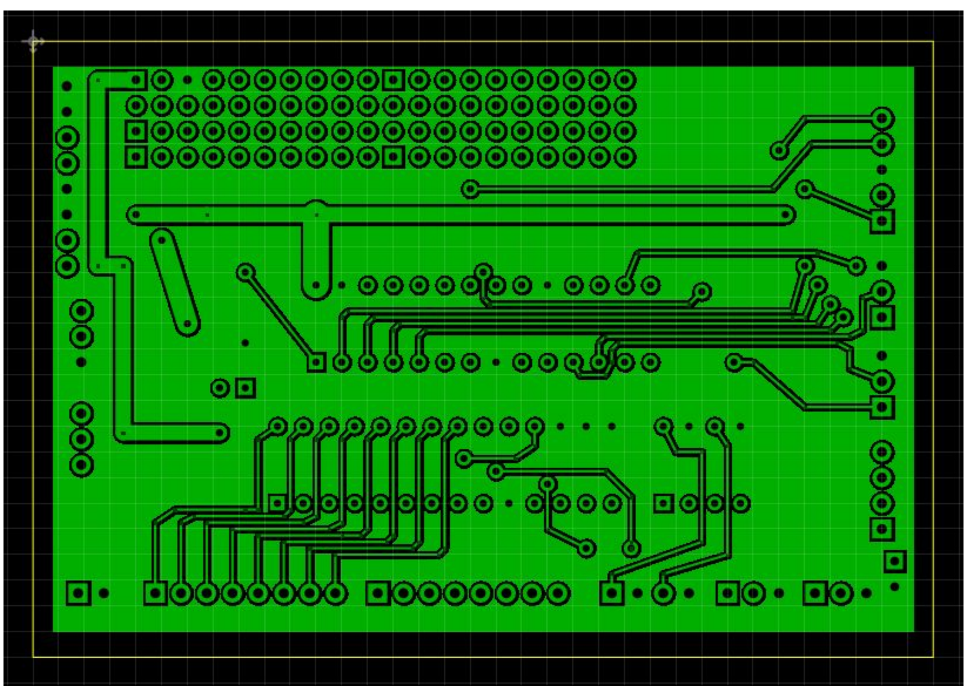


Figure A-5. PIC board Bottom Layer

Appendix B. Software Guide

The code consisting of the *PIC32Interface* python library, PIC32 firmware, module test cases are available at https://github.com/vv258/PIC-and-Pi-Interface. The final application is available at https://github.com/yk749/PIC32-and-Raspberry-Pi-Interface

	Library	**PIC32Interface**		
1	Function	CheckSum		
	Description	To calculate checksum		
	Parameters	Command	Command Bytes to be transmitted	
	Return Value	CheckSumValue	Sum of Command bytes	
	Example usage	Helper function. Not required for user		
2	Function	SendCommand		
	Description	To transmit the command over serial port		
	Parameters	Command	Command Bytes to be transmitted	
	Return Value			
	Example usage	Helper function. Not required for user		
3	Function	WriteBuffer		
	Description	To write data to Buffer memory in PIC32		
	Parameters	BufNum	Specifies the buffer number	0-3
		Data	List of data to be written	2 byte words
	Return Value			

	Example usage	`sawdata =list()`
		`for j in range(0,16):`
		`      for i in range(0,16):`
		`      sawdata.append(i*16)`
		`      sawdata.append(j)`
		
		`for j in range(15,-1,-1):`
		`      for i in range(15,-1,-1):`
		`      sawdata.append(i*16)`
		`      sawdata.append(j)`
		
		`PIC32Interface.WriteBuffer(0,sawdata)`
		
		`sinedata =list()`
		`data =list()`
		`for i in range(0,512):`
		
		`data.append(2047*math.sin(i*2*math.pi/512)+2047)`
		
		`for i in range(0,512):`
		`    MSB,LSB=divmod(int(data[i]),256)`
		`      sinedata.append(LSB)`
		`      sinedata.append(MSB)`
		
		`PIC32Interface.WriteBuffer(1,sinedata)`		

4	Function	ReadBuffer		
	Description	To read data from Buffer memory in PIC32 as single bytes		
	Parameters	BufNum	Specifies the buffer number	0-3
		DataLength	Number of 1byte words to be read	
	Return Value	Data	List containing data	
	Example usage	`ReadData=PIC32Interface.ReadBuffer(0,5)`		
5	Function	ReadBuffer2		

	Description	To read data from Buffer memory in PIC32 as two byte word		
	Parameters	BufNum	Specifies the buffer number	0-3
		DataLength	Number of 2 byte words to be read	
	Return Value	Data2	List containing data	
	Example usage	`ReadData=PIC32Interface.ReadBuffer2(3,200)`		
6	Function	SetPWMPeriod		
	Description	To Set Period for PWM as Period * (10^unit) microseconds. This is common for both channels.		
	Parameters	Period	specifies the value	0-255
		unit	specifies the power of 10	0-255
	Return Value			
	Example usage	`PIC32Interface.SetPWMPeriod(22, 3)` `#Sets period to 22 milliseconds`		
7	Function	EnablePWM1		
	Description	To Set the ON time for PWM Channel 1 as OnTime*(10^unit) microseconds and start the PWM. SetPWMPeriod should be called before calling this function		
	Parameters	Period	specifies the value	
		unit	specifies the power of 10	
	Return Value			

	Example usage	`PIC32Interface.SetPWMPeriod(22, 3)`
		`PIC32Interface.EnablePWM1(13,2)`
		`#set on time to 1.3 milliseconds`		

8	Function	EnablePWM2		

	Description	To Set the ON time for PWM Channel 2 as OnTime*(10^unit) microseconds and start the PWM. SetPWMPeriod should be called before calling this function		
	Parameters	Period	specifies the value	
		unit	specifies the power of 10	
	Return Value			

	Example usage	`PIC32Interface.SetPWMPeriod(22, 3)`
		`PIC32Interface.EnablePWM2(17,2)`
		`#set on time to 1.7 milliseconds`		

9	Function	SetDACA		
	Description	To Set DC value for DAC Channel A		
	Parameters	DacVal	specifies the DC value	0-4095
	Return Value			

	Example usage	`#Set 2V output`
		`VA=(int)(4096*2.0/3.3)`
		`PIC32Interface.SetDACA(VA)`		

10	Function	SetDACB		
	Description	To Set DC value for DAC Channel B		
	Parameters	DacVal	specifies the DC value	0-4095
	Return Value			

	Example usage	`#Set 2V output`
		`VB=(int)(4096*2.0/3.3)`
		`PIC32Interface.SetDACB(VB)`		

11	Function	ConfigureDACA		
	Description	To Configure DAC Channel A for arbitrary waveform generation		
	Parameters	BufNum	Specifies PIC buffer to be used for generating the waveform	
		Mode	0-Single burst	
			1-Continuous	

	Return Value			
	Example usage	`PIC32Interface.ConfigureDACA(0,1)` `#Setup DAC to Continuously play buffer 0`		
12	Function	ConfigureDACB		
	Description	To Configure DAC Channel B for arbitrary waveform generation		
	Parameters	BufNum	Specifies PIC buffer to be used for generating the waveform	
		Mode	0-Single burst	
			1-Continuous	
	Return Value			
	Example usage	`PIC32Interface.ConfigureDACB(1,0)` `#Setup DAC B to  play buffer 1 once`		
13	Function	ConfigureDACAB		
	Description	To Configure DAC Channel A and B for arbitrary waveform generation		
	Parameters	BufNumA	Specifies PIC buffer to be used for generating the waveform in Channel A	
		BufNumB	Specifies PIC buffer to be used for generating the waveform in Channel B	
		Mode	0-Single burst	
			1-Continuous	
	Return Value			

	Example usage	`PIC32Interface.ConfigureDACAB(0,1,1)`
		`# Play Channel A from buffer 0 and Channel B from buffer 1 continuously`
		
		`#Setup DAC B to  play buffer 1 once`		

14	Function	StartDACOutput		
	Description	To start the arbitrary waveform generation from DAC.Buffer should be written using WriteBuffer and ConfigureDACA/ConfigureDACB/ConfigureDACAB function should be called before using this function		
	Parameters	SampleFreq	Specifies sample frequency for waveform generation in kilohertz	0-255
		Samples	Specifies number of samples from buffer to use for waveform generation	
	Return Value			

	Example usage	`PIC32Interface.WriteBuffer(0,sawdata)`
		`PIC32Interface.WriteBuffer(1,sinedata)`
		`PIC32Interface.ConfigureDACAB(0,1,1)`
		`#play buffer 0 and 1 continuously`
		`PIC32Interface.StartDACOutput(10,512)`
		`#use 512 samples from buffer A and B at 10 khz for waveform generation`
		`#resultant waveform will have frequency =(10/512) khz # Play Channel A from buffer 0 and Channel B from buffer 1 continuously`		

15	Function	StopDACOutput		
	Description	To reset the DAC		
	Parameters			
	Return Value			

	Example usage	`PIC32Interface.StopDACOutput()`
		`PIC32Interface.WriteBuffer(1,sinedata)`
		`PIC32Interface.ConfigureDACAB(0,1,1)`
		`#play buffer 0 and 1 continuously`
		`PIC32Interface.StartDACOutput(10,512)`
		`#use 512 samples from buffer A and B at 10 khz for waveform generation`
		`#resultant waveform will have frequency =(10/512) khz # Play Channel A from buffer 0 and Channel B from buffer 1 continuously`		

16	Function	_SetSampleFreq_		
	Description	To set ADC sampling frequency		
	Parameters	PrescalerSetting	Specifies the sampling frequency in kilohertz	
		SampleVal	Specifies the number of samples to be acquired	
	Return Value			
	Example usage	`PIC32Interface._SetSampleFreq_( 1, 200 )` `#Acquire 200 samples at 1 khz sampling frequency`		
17	Function	_StartADC_		
	Description	To start ADC sampling. _SetSampleFreq_ should be called before calling this function		
	Parameters	Channel	Specifies the analog input channel	0-3
		Buffer	Specifies the buffer number to store the samples	0-3
	Return Value			
	Example usage	`PIC32Interface._StartADC_( 1,3 )` `#Take samples from analog channel 1 & put in buffer 3`		

18	Function	CheckBufferStatus		
	Description	To check if ADC sampling is complete		
	Parameters			
	Return Value	status	4 bit value. Each bit indicates status of corresponding PIC32 buffer	

	Example usage	`while(not((PIC32Interface.CheckBufferStatus())and 0x08) ):`
		`pass`
		`#wait till buffer 3 ADC sampling is complete`		

19	Function	WriteGPIO		
	Description	To write data to GPIO port Y		
	Parameters	data	8 bit value to write to Port Y	
	Return Value			

	Example usage	`#Turn ON all pins`
		`WriteData=255`
		`PIC32Interface.WriteGPIO(WriteData)`		

20	Function	ReadGPIO		
	Description	To read data from GPIO port Z		
	Parameters			
	Return Value	ReadVal	8 bit value read from PORT Z	
	Example usage	`ReadData=PIC32Interface.ReadGPIO()`		

Table B-1. PIC32Interface Library Functions

Appendix C. Test Plan

The following procedure may be used to test out a newly assembled board.

a. Remove the ICs from the sockets.
b. Set the power supply to 5V and turn off. Connect the J2 connector to power supply and turn On the power supply. Turn ON the switch SW2 on the board.
c. Check the output voltages on the J3 connector and turn OFF switch.
d. Place the ICs in sockets and make the following connections:
    i. Connect USB to serial converters to debug channel and RPi channel and connect to PC
    ii. Connect the programmer
    iii. Connect parallax servo motors to PWM channels
    iv. Connect DAC output channels to oscilloscope
    v. Connect analog input channels to function generator
    vi. Connect loopback connectors across GPIO Ports
    vii. Connect motor in pin to 5V out pin of J3
e. Power On the board and program the PIC with PiInterface code from repository.
f. On reset, check if *“Welcome to PIC & Pi Project Debug Window”* appears on debug serial channel.
g. Run the module tests available in the Jupyter Notebook available in the repository to verify each module.