ET 494 Engineering Technology

Senior Design Final Project Report

Xbox 360 Controller Expansion

[Open Source Hardware]

Jesse Ardonne

Computer Engineering Technology

Fall 2012

Jesse.Ardonne@selu.edu

Advisor

Dr. Cris Koutsougeras, Ph.D.

Department of CSIT

ck@selu.edu

Table of Contents

[Abstract 4](#_Toc322948580)

[Introduction 5](#_Toc322948581)

[Objectives 5](#_Toc322948582)

[Concept 5](#_Toc322948583)

[Purpose 6](#_Toc322948584)

[Design 7](#_Toc322948585)

[Design History 7](#_Toc322948586)

[Hardware 7](#_Toc322948587)

[mBed 8](#_Toc322948588)

[AD5206 8](#_Toc322948589)

[4066 9](#_Toc322948590)

[MCP23017 9](#_Toc322948591)

[LCD 16x2 Module 10](#_Toc322948592)

[Keypad [12 Button] 10](#_Toc322948593)

[Xbox 360 Controller 10](#_Toc322948594)

[Software 12](#_Toc322948595)

[mBed applications 12](#_Toc322948596)

[Computer 12](#_Toc322948597)

[Wiimote Program (C#) 12](#_Toc322948598)

[Keyboard/Mouse Program (C++) 13](#_Toc322948599)

[Integration 13](#_Toc322948600)

[Applications 16](#_Toc322948601)

[Customize Control Scheme 16](#_Toc322948602)

[Prototype 16](#_Toc322948603)

[Macro 16](#_Toc322948604)

[Exercise 17](#_Toc322948605)

[Game 17](#_Toc322948606)

[Project’s 19](#_Toc322948607)

[Compatible 19](#_Toc322948608)

[Microsoft® Kinect 19](#_Toc322948609)

[Playstation® Move 19](#_Toc322948610)

[Other 20](#_Toc322948611)

[Conclusion 21](#_Toc322948612)

[References 21](#_Toc322948613)

[Appendices 22](#_Toc322948614)

[Microcontroller 22](#_Toc322948615)

[Wiimote 22](#_Toc322948616)

[Keyboard 24](#_Toc322948617)

[Wiimote 26](#_Toc322948618)

[Keyboard/Mouse 31](#_Toc322948619)

[Body 35](#_Toc322948620)

[Schematic 36](#_Toc322948621)

[Hardware 37](#_Toc322948622)

# Abstract

Interaction with a videogame is essential to the overall experience. Motion controls are a great way to improve the interaction adding entertainment. The immediate objective of this project is to interface a microcontroller with an Xbox® 360 controller and allow input from various mediums. If a development board can be developed, it will provide the open source hardware necessary to change the way we interact with videogames. While some may want to develop new controllers for their games, others may want to customize current game controls beyond what the game permits. Some may even use it for artistic purposes at the rate realistic models and textures are being implemented. An example is using a Nintendo® Wiimote to play a First Person Shooter (FPS) game or using force sensors and accelerometers to develop a boxing game.

# Introduction

The motivation behind this project is that many gamers desire the opportunity to become more engaged in their favorite games. The gaming industry currently has its competitors contending to dominate the interactive aspect of gaming. Microsoft® offers the Kinect which has voice and bodily motion recognition. Sony® has the Playstation Move which features a camera and two remotes that have accurate motion controls for shooting and racing games. Nintendo® sells the Wii which utilizes two controllers with three accelerometers, multiple buttons, and an infrared sensor to capture movement.

Although all three of these services offer a great opportunity to become more engaged in gaming, many of the games released specifically for these devices are not the top selling games. By creating an open source platform with hardware and software, gamers with a vast range in technical knowledge can have a resource to review and expand controls to play their favorite games with any motion control they wish. Anything a gamer envisions has the potential to become reality.

## Objectives

The main objective is to develop an interface between a microcontroller and an Xbox® 360 controller. By allowing the microcontroller to accept inputs through various mediums, the microcontroller could then interpret the signals and convert them to acceptable outputs for the Xbox® 360 controller.

## Concept

The fundamental concept of this project is to use a more engaging control scheme for a game to override the generic game pad provided by Microsoft. An external sensor that could provide more engaging game play (e.g. a Wii remote) could also be connected to a computer rather than directly to a microcontroller. The computer could then echo the commands through a serial port to a microcontroller. While receiving data, microcontroller could then adjust its outputs to correspond with the commands to control an Xbox 360 controller. Other sensors may include the Microsoft Kinect, a Dance Dance Revolution pad, or any robotic sensor connected directly to the microcontroller.

## Purpose

The first of many purposes for this project is to accompany the Microsoft® XNA Game Studio. If you can program your own games, why can you not develop your own controller? Games such as Cabela’s® Big Game Hunter 2012 and Namco’s® Time Crisis 3 can be purchased along with a special controller to deliver a unique experience.

The next purpose is to customize control schemes for existing games beyond the default limit. Microsoft’s® franchise, *Halo®*, has released 6 games over the past decade. Over half of these games have seen more than one million players online at once. The default controls have you move and aim with a joystick. With this project, you can replace the controller with a Sony® Move gun controller and provide more interacting with the game.

Finally, this project was intended to spark the interest of upcoming students. Because this is an Engineering Technology major, I wanted to complete a large scale hands on project to grab the attention of younger scholars.

# Design

## Design History

The hardware chosen in this project was not what was in the original concept. The original microcontroller used was the NETduino but after finding more libraries for the mBed along with more Pulse Width Modulation (PWM) pins, the mBed was chosen early in the project. To control the 6 analog functions, the first idea was to use PWMs with a filter to provide a constant voltage. The next idea was to use a Digital to Analog (D/A) converter. The final decision was to use the AD5206 described below. This allowed fast and simple voltage changes that worked great for the project. The buttons were originally going to be normal transistors, but this was changed to the 4066 Quad Bilateral Switch when discovering the digital transistor network. This helped organize the circuit. When considering options on having more input and output pins on the microcontroller, the first idea was to use shift registers but changed to use a MCP23017 featuring an I2C bus. This allows the developer to turn 2 pins into 128 if properly implemented. Once multiple inputs were made readily available, a simple input device needed to be implemented. While the first idea was to implement a few buttons, the 12 button keypad provided an easy way to minimize wires and maximize buttons. Light Emitting Diodes (LEDs) were going to be the original indicator of what the current input is, but this turned into a 16x2 LCD module.

## Hardware

The hardware used in this project consisted of an mBed as the microcontroller, 4 quad bilateral switches, 2 I2C port expanders, and a 6 channel digital potentiometer. Additional accessories not required included an extension cable for a Playstation® 2 controller, a Playstation® 2 controller, an extension cable for a Nintendo 64® controller, a Nintendo 64® controller, a USB-a to USB mini cable, a laptop computer, and a Bluetooth dongle along with drivers.

### mBed

Based on the NXP microcontroller, the ARM Cortex M3 core runs at 96MHz with 512KB flash and 64KB RAM. The mBed has several interfaces including Ethernet, USB, CAN, SPI, I2C, and other Input/ Output (I/O). The pin layout is shown in the Appendix.

On the mBed, the first SPI connection is the AD5206 chip detailed below using pins 5, 6, and 7. Pin 8 acts as the chip select for that chip. The I2C bus uses pin 9 as the Serial Data Line (SDA) and pin 10 as the Serial Clock (SCL). Another SPI connection is used on pins 11, 12, and 13 for the Playstation® 2 controller interface. The chip select is pin 14. Pins 15 through 20 are occupied by the 16x2 LCD Module explained below. The last data pin occupied is used as a Digital I/O to poll a Nintendo 64® controller creating another interface. The ground pin is grounded with the circuit as well as the power in pin, (pin 2 or Vin) having a connection to the 5V.

### AD5206

The AD5206 Digital Potentiometer features 6 channels with 256 positions. The datasheet is included in the Appendix. While terminal resistances can be of 10kΩ, 50kΩ, and 100kΩ, the 10kΩ was chosen used in this project. Utilizing an SPI connection, interfacing only requires 3 lines between the microcontroller and IC. This is further reduced to two lines due to the chip only requiring input with no output.

The primary purpose of this chip in the project is to replace the 6 mechanical potentiometers. Once properly implemented as shown in the schematic (See Appendix), simple SPI commands such as write(address, value), with the address residing between 0 and 5 while the value lies between 0 and 255. For a full reference, see the mBed code in the Appendix.

### 4066

The 4066 Quad Bilateral Switch features a network of 4 digital transistors with a wide supply voltage range. The datasheet is included in the Appendix. The primary purpose of this chip in the project is to replace the physical button pushes of the controller. By having the enable pins controlled by the microcontroller, programs will decide when to activate the button. When programs decide when to activate a button, sensors can be used to determine it. The 16 enables between the 4 chips are connected to a single MCP23017 IC detailed below.

### MCP23017

Featuring a 16-bit remote bidirectional I/O port defaulted to input, high speed I2C interfacing, 3 hardware address pins allowing 8 devices on a single bus, and configurable interrupt ports, the MCP23017 16-Bit I/O Expander with Serial interface can be purchased for less than 3 dollars. The I2C data bus requires a 3.3kΩ pull-up resistor to 3.3V on the SDA and SCL lines. The address is set by hard wiring power or ground to the 3 address pins.

The primary purpose of this chip in the project is to expand the digital I/O pins. In the prototype build, the buttons from the Xbox® 360 controller occupied 16 of the 25 digital I/O pins on the mBed. This reduces the occupancy to 2 pins. Two of these chips are used in the project. The first one uses all 16 digital I/O pins to control one button each from the controller. The second chip acts as input for the Keypad [12 Button]. What normally would take up 23 pins on the mBed is now only using two. The address for the controller is 000 and the keypad is 001. Because mBed has a base address of 0x40, the two addresses are 0x40 and 0x42. For a full reference, see the mBed code in the Appendix.

### LCD 16x2 Module

The HD44780 LCD features a 16 character wide screen with two rows. The text is white on a blue background. Up to 8 extra characters can be created for custom glyphs on top of English / Japanese text. A potentiometer is used to adjust the contrast.

The screen occupies 6 pins of the mBed. The pin layout is depicted in the Appendix. Pins 1, 3, 5, and 16 are connected to ground with pin 3 going through the contrast potentiometer. Pins 2 and 15 are powered by 5V. Pins 4, 6, 11, 12, 13, and 14 connect to the mBed. The TextLCD library developed by Simon ford was used. This allows simple function calls such as lcd.cls() and lcd.printf(). The screen is used to show the current input that is selected by the Keypad explained below.

### Keypad [12 Button]

The keypad used in the project uses 7 output pins that can be scanned by a microcontroller. Because the buttons are setup in a matrix format, this reduces the necessary pins for button presses. Pins 3, 5, 6, and 7 require a 10kΩ pullup resistor. All 7 pins are connected as inputs to the microcontroller. Rather than occupy 7 pins on the mBed, these are connected to an MCP23017 configured as input. The schematic and code can be found in the Appendix.

### Xbox 360 Controller

The Xbox® 360 controller comes in three different models at the time of this paper. The model used is known as the Common Ground (CG) to the [www.Xbox-Scene.com](http://www.Xbox-Scene.com) community. This controller features the 16 buttons having a pullup to 3.3V. When these values are connected to ground, the on board microcontroller of the Xbox® controller reads a pressed button. The analog functions such as joysticks and triggers are supplied by a 1.52V source and can be any value in between. The traces mapped out by Ron Alexander of [www.Xbox-Scene.com](http://www.Xbox-Scene.com) and confirmed by me in this project are shown in the Appendix.

## Software

### mBed applications

The project currently has multiple separate C++ programs depending on Wiimote input, keyboard input, Nintendo® 64 input, and Playstation® 2. The aim of the design is to combine them having the keypad choose different while loops to properly adjust the outputs. The code shown in the Appendix features the Wiimote code and keyboard code. At this time, the Nintendo® 64 and Playstation® 2 program has been started but not completed.

### Computer

On the computer side, a C# application has been written for Wiimote support and a C++ program has been developed for the keyboard and mouse. Both of these aim to take in input, convert it to a byte array, and transfer the data over a serial port. The specifications are described below.

### Wiimote Program (C#)

The Wiimote program is designed to handle interrupts. Brian Peek developed a library for the Wiimote device. Although the library comes with an example program, this program did not suit the needs of this project. Emporia State University (ESU) hosts a page with projects and assignments on the school’s website. Of those assignments, a basic console template is available. Using this template, modifications were made to produce the final result shown in the Appendix.

An interrupt event handler is implemented to provide real-time updates. Everytime an update occurs, a serial port writes the updated values as well. The button presses are converted from a boolean while the Infrared (IR) values and accelerometers are converted from floats. The floats range from 0 to 1.0. When these values are multiplied by 255, they become the perfect range of a byte. Because some values are capable of spiking to infinity, an if statement prevents the data from leaving the range of a byte.

### Keyboard/Mouse Program (C++)

The keyboard program polls the necessary keys and mouse using the Allegro library. Allegro is a free download to accompany C and C++ programs with a sound, graphics, and input library. Although graphics and sound were not necessary, the input was the appealing factor. The final code is shown in the Appendix.

The code requires the Allegro library and the .NET Framework. The Allegro takes the input while the .NET Framework handles the serial output. A basic graphic is made to show the current input on the computer in real-time while it is being transmitted. After deciding on which keys to map, those keys are polled and inserted into a byte array. The mouse uses specific function calls to see how many pixels it has moved since the last update and then reset the position in the center of the screen. The cursor is also invisible on the screen. After compiling the necessary input, the byte array is copied to the .NET equivalent to be sent through the serial port. The code rest(50) is the time between each update which makes this loop occur approximately 20 times a second.

## Integration

The ultimate integration of everything in schematic form is shown in the Appendix. The first SPI connection occupying pins 5, 6, and 7 go to the AD5206. When the analog sticks or triggers are updated, this is the chip changing its digital potentiometers creating the right voltage level. Pin 6 has no connection due to the fact the AD5206 does not output anything. Pin 8 on the mBed acts as the chip select for the AD5206. Pins 9 and 10 are the I2C bus used for two MCP23017s. Pin 9, the SDA and 10, the SCL, both require a 3.3kΩ pullup resistor to the 3.3V power supply. While one MCP23017 has the address 0x00, the other has 0x01. Because the mBed has a base address of 0x40, the addresses used in the code are 0x40 and 0x42. One of the MCP23017 chips act as output writing high and low the enable pins on the 4066 chips while the other MCP23017 is an input for the keypad. Because the 4066 is a network of 4 digital transistors, the MCP23017 controls the enable while ground is always the input pin and the Xbox® controller button is the output. Because there are 16 buttons, this is a perfect setup for button pushes. Pins 11, 12, and 13 act as another SPI connection to the Playstation® 2 controller. Code was available on the mBed website to accurately obtain data from the controller. Pin 14 is the enable or attention pin for the Playstation® 2 connection. Pin 15 through 20 are used for the 16x2 LCD. The library TextLCD is used to appropriately display data. The proper configuration is shown in the schematic in the Appendix. Pin 21 is a digital I/O pin used to poll the Nintendo® 64 controller. Example code was provided by the mBed community on interpreting the values into useable data. Vin is supplied by the 5V line to have power when a USB cable is not connected. The USB cable is only required when serial data is being sent to the microcontroller.

The power supply is a 5V 1000mA going into a 2.1mm DC barrel jack. A 20 pin header is connected to the power and ground side. At the end of the 5V pin header, a 3.3V regulator is implemented with the proper capacitors to filter. The 3.3V out has a third 20 pin header allowing many devices to be implemented. A rocker switch separates the DC barrel jack from the ground pin header. This is to implement a proper way to disable power when the device is still plugged in. The Xbox® 360 controller and MCP23017 chips require 3.3V while the 4066, mBed, and AD5206 requires a 5V supply. The power line going into the AD5206 comes from the Xbox® controller board itself being the appropriate 1.52V.

# Applications

## Customize Control Scheme

The first major application of this project is to customize a control scheme beyond the manufacturers default. While some games may require a joystick to aim a gun, why not implement motion controls to move it? Racing games can have custom racing wheels replace the traditional non-interactive controls. Fighting games could be replaced with motion controls. While Kinect games continuously flood the market, very few catch the attention of long term gamers. By building this platform capable of remapping controls to different controls, motion controls, and other sensors, any game can be transformed into a more interactive experience whether its accelerometers, motion controls, or even voice control.

Prototype Controller

When developing a game, you may want to develop a controller alongside such as Cabela’s® Big Game Hunter 2012 and Namco’s® Time Crisis 3. If a developer wanted to make an interactive shooter, maybe a gun should be designed where the reticule can be anywhere on the screen rather than fixed in the middle. If a developer wanted to make a more interactive racing game, they could implement a special steering wheel accompanied by a shifting setup or even a rumble chair. This project strives to make designing a custom controller as simple as possible.

Macro Ability

Because the outputs are first sent through the computer, these can be recorded or stored. If a game was to use action sequences such as *Mortal Kombat®* or *Batman: Arkham City®*, macros could be mapped out to one button or use motion controls to activate them rather than pressing a sequence of buttons. This can go further by recording the time and inputs of the user playing a level and have it replay the level (assuming the AI was not random). Computer Science students can take this further by giving life to a second player while nobody is actually playing.

Exercise

Because most racing games utilize the right trigger of the Xbox® 360 controller, any analog input can replace it. For modern games such as *Need for Speed®*, *Burnout®*, and *Forza®*, exercise could replace the way these games are played. If someone was to max out their speed on an exercise bike creating a ‘ceiling’ variable, the speed at which they pedal could be converted into the speed in game. This is then furthered by adding the direction of the handle bars as the joystick that turns the vehicle. The end result would be someone who must pedal near their top velocity and steer becoming more interactive with the game and promote exercise.

For this setup to work either an exercise bike should be used or a properly rigged bicycle. If the back tires are safely mounted off the ground, the user could pedal while not moving. The issue is the lack of resistance. By utilizing magnets, different resistances could be set adding difficulty.

If someone was to create a new game and wanted to implement an exercise bike, the rumble motor pin on the Xbox® controller can be used. Because this is the only output of the controller, this can be treated as a pulse to give appropriate outputs. If the developer designed a track and the user goes off road, the rumble pin could send a signal to either half the speed or even better; add more resistance to the exercise bike. Power ups could be implemented adjusting all of these making resistances higher and lower creating an interactive game for people using an exercise bike.

Game Art

A recent trend in games such as *Halo®* is the ability to replay game footage. Every game is automatically recorded and you can save or upload your film. While watching the footage on your console, you can fly around the map as the camera and go watch in slow motion, fast motion, still frame, or normal speed. If you build your own map, you can take screenshots and have them automatically uploaded to your account. What started as simple screenshots has spawned many online communities dedicated to taking the best screenshots. Others, such as the administrator of [www.infectionist.com](http://www.infectionist.com) takes it further to create 3D panoramas of maps.

After finding an area of the map that seams appealing, the user must be precise in measuring the snapshot. Starting with looking all the way down, you take a picture, and rotate 30º until you reach the beginning. You then tilt the camera up 15º and repeat the 12 rotating screenshots. At the end, you will have 156 screenshots. The next step involves stitching the images in a spherical panorama format and turning them into an Adobe® Flash file. While this can be extremely tedious to do by yourself, the option to preprogram a macro is available through this project. This means that you can find the spot you like, whether it is a still area or a big action scene, still frame the environment, and run the macro. This cuts significant time for the artists.

Project’s Future

Compatible Accessories

### Microsoft® Kinect

Kinect is a motion sensing input device developed by Microsoft® for the Xbox® 360 and Windows PCs. Through a natural user interface such as gestures and voice commands, the Kinect enables users to interact with the Xbox® 360. The internal software technology was developed by Rare® based on range camera technology by PrimeSense®.

For a game to use Kinect controls, the manufacturer must have the support available. Thanks to Microsoft® for releasing the Kinect for Windows SDK, you can now connect the Kinect to your computer and receive useable input. Using that input, data can be converted to replace the controller in a current game without Kinect support.

The limitation is that it may only substitute a button press or analog function. For example, if you are playing a sword fighting game and you want to be able to swing the sword by waving your hand, that is possible. What is not possible (with this project at least) is to have the character mimic body motion. This must be coded into the game itself, not the controls.

### Playstation® Move

The Playstation® Move is exclusive to the Playstation 3® platform acting as a motion-sensing gaming controller. Similar to the Nintendo® Wii, it is based on a handheld motion controller wand and a sensing element. In this case, it is the PlayStation Eye® camera accompanied by inertial sensors. While this is obviously not used for any Xbox 360® title, now it can be with this project.

For $100, you can download Move.Me, a required app from the Playstation Network to take in user data. The Playstation® Move SDK can then be downloaded on the computer. While the Playstation 3® is powered on with the Move connected, calls can be made over a Local Area Network (LAN) to get the data on the computer. Once the data is on the computer, you can translate it into your desired controls using the project.

### Other

Other sensors may be a Dance Dance Revolution (DDR) pad which has 8 arrows. These 8 direction buttons could be directly translated to the analog stick controlling movement. The result would be the direction you step is the direction you move. A project in the United Kingdom has used a 360º Treadmill which allows a person to walk freely and while they slowly make their way to the center, the movement is translated into the game. Another feature is a screen that is 360º. Projectors cast the image onto the screen in front of the player. This means that if the player turns 90º left, not only does he turn 90º left in the game, but the screen still remains in front of him.

The project in this paper is great start to creating a greater experience like this simulator. By combining a pointing device (Wiimote or Move), a projector can have a circuit built where it points in front of the player. A group of engineering students could complete a project using a projector, a mechanical device for it to rotate about, and imagination. Using a Kinect to see if you are crouching or jumping, a lot of interaction can be added into already released popular games.

# Conclusion

In conclusion, this project has multiple applications as well as a bright future. It appeals to the younger generation growing up in a technology oriented time, promotes exercise and interactivity with video games, and provides an expandable platform for hobbyists and engineers to better. This project shows Bluetooth devices, USB devices, and on board sensors can be used to control the Xbox® 360 controller. Macro abilities can be implemented adding projects for computer scientists while the simulator environments can be developed by engineers. This project was merely a beginning to something that can hopefully be adapted from console to console in the future inspiring project after project and student after student.

# References

[www.mBed.org](http://www.mBed.org)

[www.netduino.com](http://www.netduino.com)

[www.arduino.cc](http://www.arduino.cc)

[www.analogdevices.com](http://www.analogdevices.com)

[www.xbox-scene.com](http://www.xbox-scene.com)

[www.sparkfun.com](http://www.sparkfun.com)

[www.adafruit.com](http://www.adafruit.com)

[www.microsoft.com](http://www.microsoft.com)

[www.allegro.cc](http://www.allegro.cc)

<http://brannigan.emporia.edu/projects/WII/>

[www.infectionist.com](http://www.infectionist.com)

[www.digikey.com](http://www.digikey.com)

# Appendices

Microcontroller Code

### **Wiimote**

#include "mbed.h"

#include "MCP23017.h"

#include "TextLCD.h"

DigitalOut led1(LED1);

DigitalOut led2(LED2);

DigitalOut led3(LED3);

DigitalOut led4(LED4);

//Declare Chip Select Pin

DigitalOut cs(p8);

TextLCD lcd(p15, p16, p17, p18, p19, p20); // rs, e, d4-d7

MCP23017 x = MCP23017( p9, p10, 0x42);

SPI jst(p5, p6, p7); // mosi, miso, sclk

//Serial Port Connection

Serial pc(USBTX, USBRX);

//Prototype for Joystick & Trigger Values

void digitalPotWrite(int,int);

int main() {

pc.baud(115200);

x.config(0,0,0);

int buffer[23];

int i;

digitalPotWrite(0, 127); // 0 is right 255 is left

digitalPotWrite(1, 127); //0 is back 255 is forward

digitalPotWrite(2, 0);

digitalPotWrite(3, 127); // 0 is right 255 is left

digitalPotWrite(4, 127); // 0 is down 255 is up

digitalPotWrite(5, 0);

lcd.printf("Test2!");

while(1){

if (pc.readable()){

for (i= 0; i < 23; i++){

buffer[i] = (int)pc.getc();

}

/\*[6] JoyX \*/ digitalPotWrite (0, 255-buffer[6]);

/\*[7] JoyY \*/ digitalPotWrite (1,buffer[7]);

/\*[21] xIR \*/ digitalPotWrite (3,buffer[21]);

/\*[22] yIR \*/ digitalPotWrite (4,255-buffer[22]);

/\*[9] B \*/ digitalPotWrite (5, buffer[9])\*250);

/\*[5] nunAccZ \*/ if (buffer[5] < 128)

digitalPotWrite (2,250);

else

digitalPotWrite (2, 0);

buffer[8] ? x.write\_bit(1,15) : x.write\_bit(0,15);

buffer[12] ? x.write\_bit(1,14) : x.write\_bit(0,14);

buffer[14] ? x.write\_bit(1,13) : x.write\_bit(0,13);

buffer[15] ? x.write\_bit(1,12) : x.write\_bit(0,12);

buffer[20] ? x.write\_bit(1,11) : x.write\_bit(0,11);

(buffer[2] < 128) ? x.write\_bit(1,10) : x.write\_bit(0,10);

buffer[13] ? x.write\_bit(1, 9) : x.write\_bit(0, 9);

buffer[17] ? x.write\_bit(1, 8) : x.write\_bit(0, 8);

/\* Dpad Buttons Not Mapped

buffer[] ? x.write\_bit(1, 7) : x.write\_bit(0, 7);

buffer[] ? x.write\_bit(1, 6) : x.write\_bit(0, 6);

buffer[] ? x.write\_bit(1, 5) : x.write\_bit(0, 5);

buffer[] ? x.write\_bit(1, 4) : x.write\_bit(0, 4);

\*/ // Sync Button not Mapped

//buffer[] ? x.write\_bit(1, 3) : x.write\_bit(0, 3); buffer[16] ? x.write\_bit(1, 2) : x.write\_bit(0, 2);

buffer[10] ? x.write\_bit(1, 1) : x.write\_bit(0, 1);

buffer[18] ? x.write\_bit(1, 0) : x.write\_bit(0, 0);

}

}

}

//Function for controlling Joysticks and Trigger Analog Signals

void digitalPotWrite(int address, int value) {

// Take the CS pin low to select the chip:

cs=0;

// Send in the address and value via SPI:

jst.write(address);

jst.write(value);

// Take the CS pin high to de-select the chip:

cs=1;

}

### Keyboard

#include "mbed.h"

#include "MCP23017.h"

#include "TextLCD.h"

DigitalOut led1(LED1);

DigitalOut led2(LED2);

DigitalOut led3(LED3);

DigitalOut led4(LED4);

//Declare Chip Select Pin

DigitalOut cs(p8);

TextLCD lcd(p15, p16, p17, p18, p19, p20); // rs, e, d4-d7

MCP23017 x = MCP23017( p9, p10, 0x42);

SPI jst(p5, p6, p7); // mosi, miso, sclk

//Serial Port Connection

Serial pc(USBTX, USBRX);

//Prototype for Joystick & Trigger Values

void digitalPotWrite(int,int);

int main() {

pc.baud(115200);

x.config(0,0,0);

int buffer[22];

int i;

digitalPotWrite(0, 127); // 0 is right 255 is left

digitalPotWrite(1, 127); //0 is back 255 is forward

digitalPotWrite(2, 0);

digitalPotWrite(3, 127); // 0 is right 255 is left

digitalPotWrite(4, 127); // 0 is down 255 is up

digitalPotWrite(5, 0);

lcd.printf("WOO!");

while(1){

if (pc.readable()){

for (i= 0; i < 22; i++){

buffer[i] = (int)pc.getc();

}

/\*[0] LSX -> A/D \*/ digitalPotWrite (1, buffer[0]);

/\*[1] LSY -> W/S \*/ digitalPotWrite (0, 255-buffer[1]);

/\*[2] RSX -> MoX \*/ digitalPotWrite (3, 255-buffer[2]);

/\*[3] RSY -> MoY \*/ digitalPotWrite (4, 255-buffer[3]);

/\*[4] RT -> RCl \*/ digitalPotWrite (5,buffer[4]);

/\*[5] LT -> LCl \*/ digitalPotWrite (2,buffer[5]);

buffer[6] ? x.write\_bit(1, 15) : x.write\_bit(0, 15);

buffer[7] ? x.write\_bit(1, 14) : x.write\_bit(0, 14);

buffer[8] ? x.write\_bit(1, 13) : x.write\_bit(0, 13);

buffer[9] ? x.write\_bit(1, 12) : x.write\_bit(0, 12);

buffer[10] ? x.write\_bit(1, 11) : x.write\_bit(0, 11);

buffer[11] ? x.write\_bit(1, 10) : x.write\_bit(0, 10);

buffer[12] ? x.write\_bit(1, 9) : x.write\_bit(0, 9);

buffer[13] ? x.write\_bit(1, 8) : x.write\_bit(0, 8);

buffer[14] ? x.write\_bit(1, 0) : x.write\_bit(0, 0);

buffer[15] ? x.write\_bit(1, 1) : x.write\_bit(0, 1);

buffer[16] ? x.write\_bit(1, 2) : x.write\_bit(0, 2);

buffer[17] ? x.write\_bit(1, 3) : x.write\_bit(0, 3);

buffer[18] ? x.write\_bit(1, 4) : x.write\_bit(0, 4);

buffer[19] ? x.write\_bit(1, 5) : x.write\_bit(0, 5);

buffer[20] ? x.write\_bit(1, 6) : x.write\_bit(0, 6);

buffer[6] ? x.write\_bit(1, 7) : x.write\_bit(0, 7);

}

}

}

//Function for controlling Joysticks and Trigger Analog Signals

void digitalPotWrite(int address, int value) {

// Take the CS pin low to select the chip:

cs=0;

// Send in the address and value via SPI:

jst.write(address);

jst.write(value);

// Take the CS pin high to de-select the chip:

cs=1;

}

Wiimote Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using WiimoteLib;

using System.IO.Ports;

#region CollectWiimotes Class

// Contains Dictionary to map Guids to a Class Instance

// Contains our Wiimote Collection to manage all of our connected wiimotes

public partial class CollectWiimotes

{

// Dictionary holds Guids and class instances

// Allows us to easily access an individual wiimote

private static Dictionary<Guid, WiimoteInfo> mWiimoteMap = new Dictionary<Guid, WiimoteInfo>();

// WiiMoteCollection is a class from the DLL that stores all

// connected wiimotes

private static WiimoteCollection mWC = new WiimoteCollection();

// Standard Get/Set methods for our Dictionary/Collection

public static Dictionary<Guid, WiimoteInfo> wmMap

{

get { return mWiimoteMap; }

set { mWiimoteMap = value; }

}

public static WiimoteCollection WiiCollect

{

get { return mWC; }

set { mWC = value; }

}

}

#endregion

#region WiimoteInfo Class

// Contains procedures to read from and write to Wiimotes

public partial class WiimoteInfo

{

private Wiimote mWiimote;

public int WiimoteLED;

public SerialPort sp = new SerialPort("COM3", 115200, Parity.None, 8, StopBits.One);

byte[] buff = new byte[23];

double temp;

byte xAccel, yAccel, zAccel, xNun, yNun, zNun, xJoy, yJoy, ABut, BBut, OneBut, TwoBut, UpBut, DownBut, LeftBut, RightBut, PlusBut, MinusBut, HomeBut, CBut, ZBut, xIR, yIR;

// semaphore used to prevent race condition in console display

static private Semaphore updateSemaphore = new Semaphore(1, 1);

//Default Constructor

public WiimoteInfo()

{

}

// Assignment constructor

// Is passed a Wiimote object and assigns it to the instance in the WiimoteInfo class

public WiimoteInfo(Wiimote wm)

: this()

{

mWiimote = wm;

sp.Open();

}

// Used for updating the wiimote itself

// This is where you put output statements to see changes in the wiimote state

public void UpdateState(WiimoteChangedEventArgs args)

{

WiimoteState ws = args.WiimoteState;

updateSemaphore.WaitOne();

Console.SetCursorPosition(0, this.WiimoteLED);

if ((temp = ws.AccelState.Values.X) < 1 && temp > -1)

buff[0] = xAccel = Convert.ToByte(temp \* 128 + 127);

if ((temp = ws.AccelState.Values.Y) < 1 && temp > -1)

buff[1] = yAccel = Convert.ToByte(temp \* 128 + 127);

if ((temp = ws.AccelState.Values.Z) < 1 && temp > -1)

buff[2] = zAccel = Convert.ToByte(temp \* 128 + 127);

if ((temp = ws.NunchukState.AccelState.Values.X) < 1 && temp > -1)

buff[3] = xNun = Convert.ToByte(temp \* 128 + 127);

if ((temp = ws.NunchukState.AccelState.Values.Y) < 1 && temp > -1)

buff[4] = yNun = Convert.ToByte(temp \* 128 + 127);

if ((temp = ws.NunchukState.AccelState.Values.Z) < 1 && temp > -1)

buff[5] = zNun = Convert.ToByte(temp \* 128 + 127);

if ((temp = ws.NunchukState.Joystick.X) < 0.5 && temp > -0.5)

buff[6] = xJoy = Convert.ToByte(temp \* 256 + 127);

if ((temp = ws.NunchukState.Joystick.Y) < 0.5 && temp > -0.5)

buff[7] = yJoy = Convert.ToByte(temp \* 256 + 127);

buff[8] = ABut = Convert.ToByte(ws.ButtonState.A);

buff[9] = BBut = Convert.ToByte(ws.ButtonState.B);

buff[10] = OneBut = Convert.ToByte(ws.ButtonState.One);

buff[11] = TwoBut = Convert.ToByte(ws.ButtonState.Two);

buff[12] = UpBut = Convert.ToByte(ws.ButtonState.Up);

buff[13] = DownBut = Convert.ToByte(ws.ButtonState.Down);

buff[14] = LeftBut = Convert.ToByte(ws.ButtonState.Left);

buff[15] = RightBut = Convert.ToByte(ws.ButtonState.Right);

buff[16] = PlusBut = Convert.ToByte(ws.ButtonState.Plus);

buff[17] = MinusBut = Convert.ToByte(ws.ButtonState.Minus);

buff[18] = HomeBut = Convert.ToByte(ws.ButtonState.Home);

buff[19] = CBut = Convert.ToByte(ws.NunchukState.C);

buff[20] = ZBut = Convert.ToByte(ws.NunchukState.Z);

if (ws.IRState.IRSensors[0].Found && ws.IRState.IRSensors[1].Found)

{

buff[21] = xIR = Convert.ToByte(ws.IRState.IRSensors[0].Position.X\*255);

buff[22] = yIR = Convert.ToByte(ws.IRState.IRSensors[0].Position.Y\*255);

}

Console.Clear();

Console.WriteLine("\n\n\n\n\n" +

xAccel + " " + yAccel + " " + zAccel + "\n" +

xNun + " " + yNun + " " + zNun + "\n " +

xJoy + " " + yJoy + " " + "\n" +

ABut + BBut + OneBut + TwoBut + UpBut + DownBut + LeftBut +

RightBut + PlusBut + MinusBut + HomeBut + CBut + ZBut + "\n" +

xIR + " " + yIR + "\n");

sp.Write(buff, 0, 23);

updateSemaphore.Release();

}

// Used for updating the extensions (if any are plugged in)

public void UpdateExtension(WiimoteExtensionChangedEventArgs args)

{

updateSemaphore.WaitOne();

Console.SetCursorPosition(40, this.WiimoteLED);

Console.WriteLine(" ");

Console.SetCursorPosition(40, this.WiimoteLED);

Console.WriteLine(args.ExtensionType.ToString());

updateSemaphore.Release();

}

// get/set methods for the class

public Wiimote Wiimote

{

get { return mWiimote; }

set { mWiimote = value; }

}

}

#endregion

#region HelloWorld Console Application

namespace WiiRemote\_HelloWorld\_Console\_Application

{

class Program

{

static void Main(string[] args)

{

int wiimoteIndex = 0; //Used to identify the wiimotes.

// Try and find all wiimotes. Throw an error if we can't find any.

try

{

CollectWiimotes.WiiCollect.FindAllWiimotes();

}

catch (WiimoteNotFoundException ex)

{

Console.WriteLine(ex.Message);

Console.WriteLine("Wiimote not found");

}

catch (WiimoteException ex)

{

Console.WriteLine(ex.Message);

Console.WriteLine("Wiimote Error");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

Console.WriteLine("Unknown Error In Acquiring Wiimote");

}

// Run through each wiimote we found in the HID list and:

// Instantiate a new Class

// Put its ID into the dictionary

// Connects Wiimote

// Associate the LEDE number with the instance

// Changes LED to indicate unit number

foreach (Wiimote wm in CollectWiimotes.WiiCollect)

{

Console.Clear();

// Create a new class instance for each wiimote we find in HID list

WiimoteInfo wi = new WiimoteInfo(wm);

// Place our wiimote's ID into the dictionary.

CollectWiimotes.wmMap[wm.ID] = wi;

wiimoteIndex++;

// Set the LED count

wi.WiimoteLED = wiimoteIndex;

// Connect it and tell it which functions to call when the wiimote updates state

wm.WiimoteChanged += wm\_WiimoteChanged; // add functor

wm.WiimoteExtensionChanged += wm\_WiimoteExtensionChanged;

wm.Connect();

// Set the acquisition rate for the wiimote

if (wm.WiimoteState.ExtensionType != ExtensionType.BalanceBoard)

wm.SetReportType(InputReport.IRExtensionAccel, WiimoteLib.IRSensitivity.Maximum, true);

// Set LEDs so we know it is connected

wm.SetLEDs(wi.WiimoteLED);

Console.SetCursorPosition(0, wi.WiimoteLED);

Console.WriteLine("Wiimote " + wi.WiimoteLED.ToString() + " Connected!");

Console.SetCursorPosition(0, wi.WiimoteLED + 1);

Console.WriteLine("Press escape to exit");

}

// Hang around and display the battery value (in UpdateState)

ConsoleKeyInfo info = Console.ReadKey();

while (info.Key != ConsoleKey.Escape)

{

info = Console.ReadKey();

}

Console.Clear();

}

// mathod that gets called any time the wiimote changes state

private static void wm\_WiimoteChanged(object sender, WiimoteChangedEventArgs e)

{

WiimoteInfo wi = CollectWiimotes.wmMap[((Wiimote)sender).ID];

wi.UpdateState(e);

}

// Invoked when extensions are attached

private static void wm\_WiimoteExtensionChanged(object sender, WiimoteExtensionChangedEventArgs e)

{

// find the control for this Wiimote

WiimoteInfo wi = CollectWiimotes.wmMap[((Wiimote)sender).ID];

wi.UpdateExtension(e);

if (e.Inserted)

((Wiimote)sender).SetReportType(InputReport.IRExtensionAccel, true);

else

((Wiimote)sender).SetReportType(InputReport.IRAccel, true);

}

}

}

#endregion

Keyboard/Mouse Code

#using <System.dll>

using namespace System;

using namespace System::IO::Ports;

#define USE\_CONSOLE

#include <cstdlib>

#include <iostream>

#include <allegro.h>

#define SENSITIVITY 1

#define MOUSE\_SPEED 2

BITMAP \*buffer;

void composeOutput();

int getMouseSpeed(char c);

void resetMouse();

void getControls();

Byte controls[22];

int \*controlPtr;

int xAim, yAim;

int \*xAimPtr = &xAim;

int \*yAimPtr = &yAim;

char mousePo[256];

int main(void)

{

allegro\_init();

install\_keyboard();

install\_mouse();

SerialPort^ sp = gcnew SerialPort("COM3", 115200);

sp->Parity = Parity::None;

sp->DataBits = 8;

sp->StopBits = StopBits::One;

sp->Open();

buffer = create\_bitmap(640,480);

clear\_to\_color(buffer, makecol(255, 255, 255));

if (set\_gfx\_mode(GFX\_SAFE, 640, 480, 0, 0) != 0) {

set\_gfx\_mode(GFX\_TEXT, 0, 0, 0, 0);

allegro\_message("Unable to set any graphic mode\n%s\n", allegro\_error);

return 1;

}

int x;

array<Byte>^ controls2 = gcnew array<Byte>(22);

set\_mouse\_speed(MOUSE\_SPEED,MOUSE\_SPEED);

while(!key[KEY\_1]){

get\_mouse\_mickeys(xAimPtr,yAimPtr);

getControls();

resetMouse();

composeOutput();

acquire\_screen();

blit(buffer,screen,0,0,0,0,640,480);

release\_screen();

for (x = 0; x < 22; x++){

controls2[x] = controls[x];

}

sp->Write(controls2, 0, 22);

rest(50);

clear\_bitmap(buffer);

clear\_to\_color(buffer, makecol(255,255,255));

}

sp->Close();

return 1;

}

END\_OF\_MAIN()

void composeOutput(){

sprintf\_s(mousePo, "W/S: %3d A/D: %3d RSX: %3d RSY: %3d",

controls[0], controls[1], controls[2], controls[3]);

textout\_centre\_ex(buffer, font, mousePo, SCREEN\_W/2, SCREEN\_H/6, makecol(0,0,0),-1);

sprintf\_s(mousePo, "R/T: %3d L/T: %3d",

controls[4], controls[5]);

textout\_centre\_ex(buffer, font, mousePo, SCREEN\_W/2, SCREEN\_H/4, makecol(0,0,0),-1);

sprintf\_s(mousePo, " A: %d B: %d X: %d Y: %d",

controls[6], controls[7], controls[8], controls[9]);

textout\_centre\_ex(buffer, font, mousePo, SCREEN\_W/2, SCREEN\_H/3, makecol(0,0,0),-1);

sprintf\_s(mousePo, " LB: %d RB: %d LSC: %d RSC: %d",

controls[10], controls[11], controls[12], controls[13]);

textout\_centre\_ex(buffer, font, mousePo, SCREEN\_W/2, SCREEN\_H/2, makecol(0,0,0),-1);

sprintf\_s(mousePo, "GUI: %d STR: %d BCK: %d SYN: %d",

controls[14], controls[15], controls[16], controls[17]);

textout\_centre\_ex(buffer, font, mousePo, SCREEN\_W/2, SCREEN\_H/3\*2, makecol(0,0,0),-1);

sprintf\_s(mousePo, " Up: %d DWN: %d LFT: %d RGT: %d",

controls[18], controls[19], controls[20], controls[21]);

textout\_centre\_ex(buffer, font, mousePo, SCREEN\_W/2, SCREEN\_H/6\*5, makecol(0,0,0),-1);

}

void resetMouse(){

position\_mouse(SCREEN\_W/2, SCREEN\_H/2);

//get\_mouse\_mickeys(xAimPtr,yAimPtr);

}

int getMouseSpeed(char c){

//xAim \*= 5;

//yAim \*= 5;

if(\*xAimPtr > 127)

\*xAimPtr = 128;

if(\*xAimPtr < -128)

\*xAimPtr = -127;

if(yAim > 127)

yAim = 128;

if(yAim < -128)

yAim = -127;

if (c == 'x'){

return (\*xAimPtr+127);

}

if (c == 'y'){

return (yAim+127);

}

}

void getControls(){

if (key[KEY\_W]){

controls[0] = 255;

}

else if (key[KEY\_S]){

controls[0] = 0;

}

else controls[0] = 127;

if (key[KEY\_D]){

controls[1] = 255;

}

else if (key[KEY\_A]){

controls[1] = 0;

}

else controls[1] = 127;

controls[2] = getMouseSpeed('x');

controls[3] = getMouseSpeed('y');

controls[4] = (mouse\_b & 1) \* 255;

controls[5] = (mouse\_b & 2) \* 127;

controls[6] = (key[KEY\_SPACE] & 1);

controls[7] = (key[KEY\_G] & 1);

controls[8] = (key[KEY\_R] || key[KEY\_E]);

controls[9] = (key[KEY\_Q] & 1);

controls[10] = (key[KEY\_LSHIFT] & 1);

controls[11] = (key[KEY\_F] & 1);

controls[12] = (key[KEY\_LCONTROL] & 1);

controls[13] = ((mouse\_b & 4)\*1/4 || key[KEY\_Z]);

controls[14] = (key[KEY\_TILDE] & 1);

controls[15] = (key[KEY\_ESC] & 1);

controls[16] = (key[KEY\_TAB] & 1);

controls[17] = (key[KEY\_RCONTROL] & 1);

controls[18] = (key[KEY\_UP] & 1);

controls[19] = (key[KEY\_DOWN] & 1);

controls[20] = (key[KEY\_LEFT] & 1);

controls[21] = (key[KEY\_RIGHT] & 1);

}

/\*

controls[0] WS -> Left Stick X

controls[1] AD -> Left Stick Y

controls[2] mX -> Right Stick X

controls[3] mY -> Right Stick Y

controls[4] RightClick -> RightTrigger

controls[5] LeftClick -> Left Trigger

controls[6] Space -> A

controls[7] G -> B

controls[8] R -> X

controls[9] Q -> Y

controls[10] SHIFT -> LB

controls[11] F -> RB

controls[12] CTRL -> Left Stick Click

controls[13] MiddleClick -> RightStickClick

controls[14] ~ -> Guide

controls[15] ESC -> Start

controls[16] TAB -> Back

controls[17] 0 -> Sync

controls[18] ^ - ^

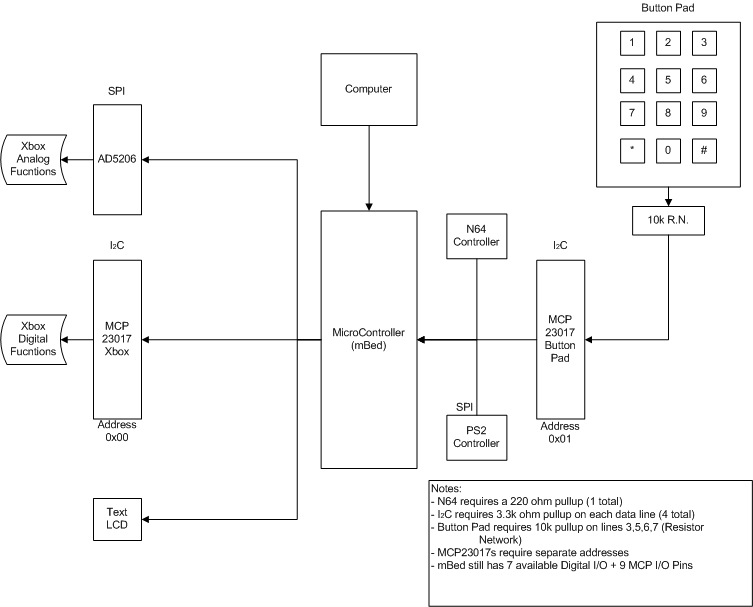
controls[19] v - v

controls[20] < - <

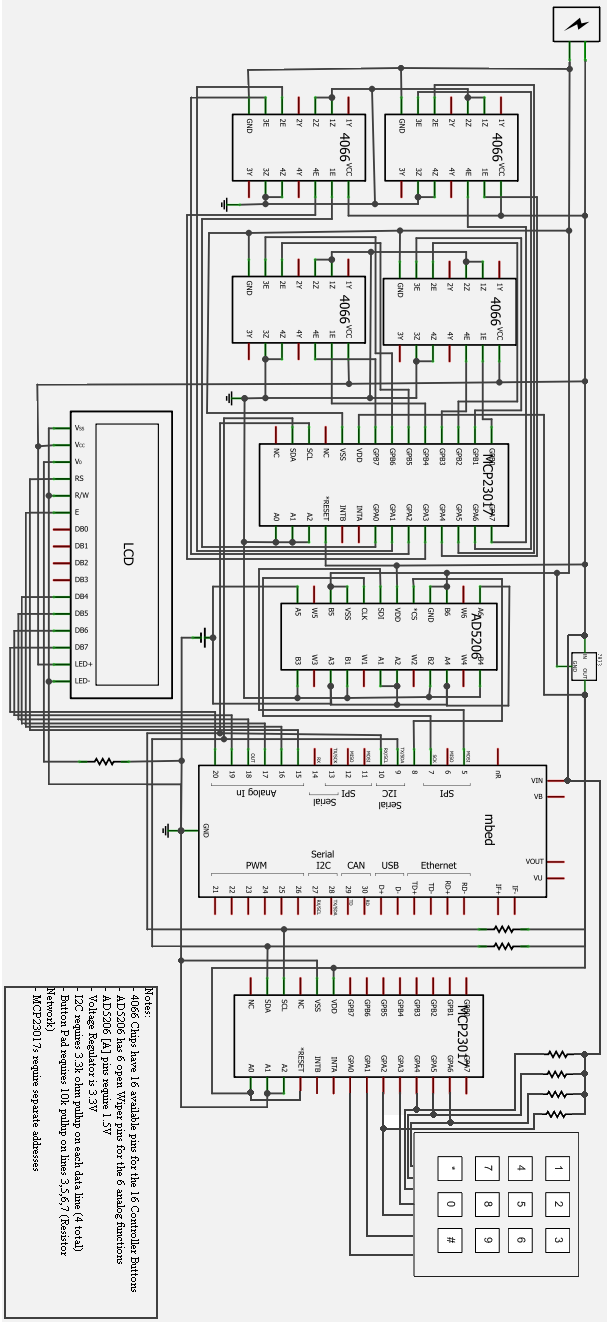
controls[21] > - >

\*/

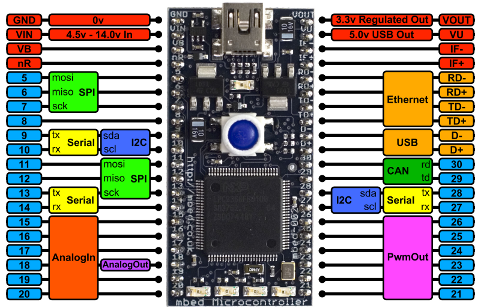
Body Diagram

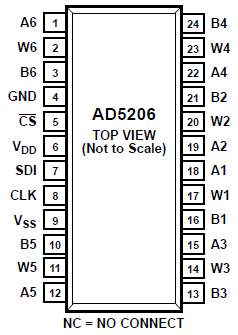
****

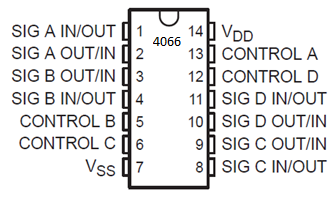
Schematic

****

Hardware Specs





****

