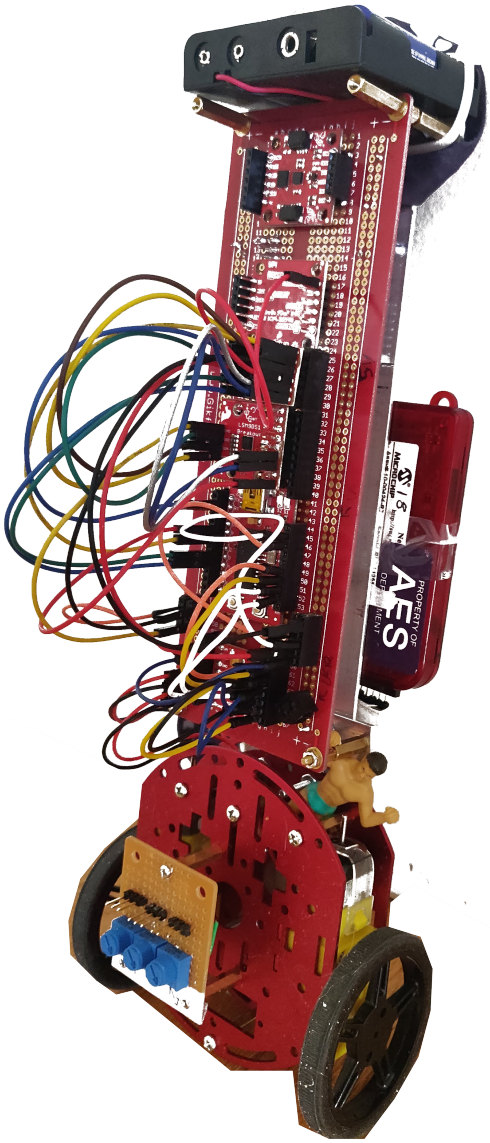
ASEN 5519 Final Project
Rio McMahon
Fall 2019

“Tipsy”



OVERVIEW

My final project for ASEN 5519 is a PIC18F2553 based inverted pendulum robot. Control is handled by an empirically tuned PID controller based on input from an IMU with sensor fusion via a complementary filter. “Tipsy”, as she’s affectionately known, is designed to stand up by actuating two DC motors based on output from an empirically tuned PID controller. This goal has been semi-completed and Tipsy is able to achieve/maintain some balance.

The project can be broken down into several major components, which are discussed in detail throughout the following report. Primary project components include:

- IMU Communication via SPI
- Sensor Fusion of IMU Data
- PID Control
- DC Motor Control

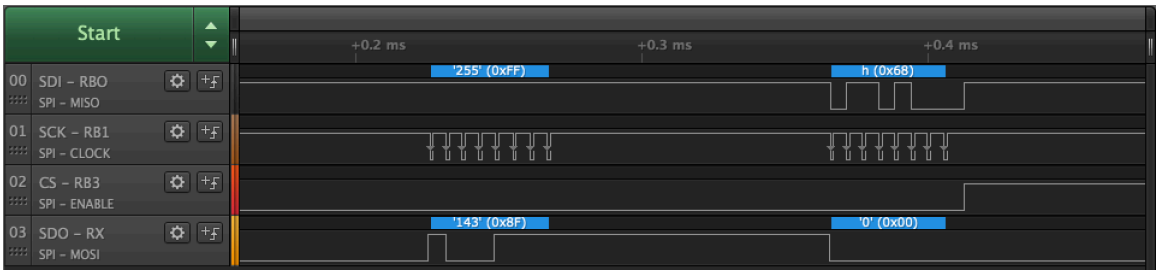
Functional hardware block diagrams, wiring schematics, materials used, software flowcharts, software documentation, datasheets, and project references can be found in their respective appendices at the end of the report.

MAJOR PROJECT COMPONENTS

IMU Communication via SPI

An LSM9DS1 IMU was used for the estimation of the angle of departure from a vertical condition. To interface with the IMU, the SPI communication protocol was used. This was chosen for its simplicity and faster rate of communication compared to the I2C protocol.

The project uses a “bit banged” software implementation of SPI as opposed to using the built in MSSP module that is included in the PIC18F2553. This approach was chosen due to hardware implementation difficulties discussed in the results section. The software SPI implementation uses timing delays and GPIO pins to generate a waveform that was able to successfully communicate with the IMU as shown in the image below. Notice that the timing of an equivalent SCK waveform is slightly irregular, however when the “WHO_AM_I” register is read from, the IMU correctly responds with its ID code 0x68.



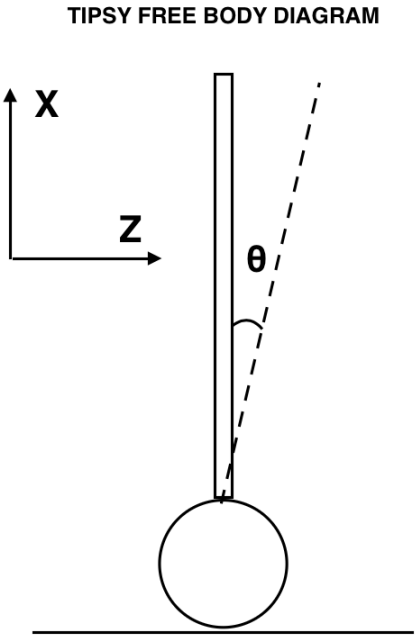
Besides the software implementation of SPI, communication with the IMU is relatively straightforward. The LSM9DS1 data sheet has a well documented list of read/write registers to access IMU data and configure the sensors. Data is read/written in a 16-bit operation which requires two SPI communication cycles. Read operations are accomplished by sending the first byte of data with the MSB = 1, whereas write operations are sent with MSB = 0. The second byte transmitted is the data being written (or dummy data for read operations). The IMU response is also transmitted on the second byte transmission.

Sensor Fusion of IMU Data

Once the data was received from the IMU, Tipsy uses a complementary filter to fuse the data from the gyroscope and the accelerometers (Dwyer 2014). The reason a filter is used is due to the high levels of noise in the accelerometer data and the gyroscopes tendency to drift. The complementary filter combines the data to reduce the contribution of noise to the state estimation, while also using the accelerometer data as a reference point to ameliorate gyroscope drift effects. The formula is:

$$\theta_k = \alpha\left(\theta_{k-1} + \dot{\theta} * dt\right) + (1-\alpha)\left(\arctan\left(\frac{Accel_x}{Accel_z}\right)\right)$$

where θ_k = estimated angle from vertical, degrees
α = empirical coefficient to weight gyroscope data/accelerometer data (implemented as 0.99)
$\dot{\theta}$ = gyroscope output, degrees per second
dt = time step between k and k+1, 10ms
$Accel_x$ = x axis accelerometer output, ratio compared to 1g
$Accel_z$ = z axis accelerometer output, ratio compared to 1g



From this, the angle of departure from vertical (theta) was measured for implementation in the PID controller.

PID Control

The estimate theta is used by a PID controller to determine inputs to the motor driver (Dwyer 2014). The PID controller uses empirically tuned K_i, K_P, and K_D determined through iterative testing. The PID control formulas are:

$$\delta_{D,k} = K_D(e_k - e_{k-1})$$
$$\delta_{P,k} = K_P e_k$$
$$\delta_{I,k} = \delta_{I,k-1} + K_I e_k$$
$$PID = \delta_{I,k} + \delta_{D,k} + \delta_{P,k}$$

e_k = error (0 – theta) at time k
K = empirically tuned PID factors
PID = PID control output used to actuate motors

The control algorithm was highly empirical and most of the factors (especially K and PID) were scaled using various scale factors to reach stability in balancing. Additionally, the angle the PID controller was attempting to minimize error against was used as an additional tuning knob; this was discovered because Tipsy had a tendency to fall backwards, likely due to mass eccentricity from the overall center of gravity. Inspiration for the selection of tuning factors came from several sources (Dwyer 2014, Control Tutorials). Once processed, the PID variable was passed to the motor driver function to actuate the motors to combat any instability.

DC Motor Control

Once the appropriate PID output is determined, Tipsy implements the control feedback by actuating two 5v DC motors to combat any instability. This actuation is accomplished by generating a PWM wave with a duty cycle to match the PID output. The underlying software uses an interrupt service routine (ISR) to handle the timing requirements of the PWM wave.

The PWM wave is sent from the PIC18F2553 to a TB6612FNG motor driver control chip which accepts the PWM wave and motor select signals to drive the two DC motors. The motor driver chip has several different pins which can be driven high or low to achieve motor control. Different variations can drive wheels in CW or CCW orientations. The different pin variations are in the documentation from Sparkfun (shown in the table below). Because Tipsy is approximating a 2 dimensional system, both motors received the same PWM/orientation commands.

In1	In2	PWM	Out1	Out2	Mode
H	H	H/L	L	L	Short brake
L	H	H	L	H	CCW
L	H	L	L	L	Short brake
H	L	H	H	L	CW
H	L	L	L	L	Short brake
L	L	H	OFF	OFF	Stop

RESULTS

Summary

Overall, Tipsy is occasionally able to balance for short periods of time. There is a fair amount of jitter due to the low quality motors and she has a tendency to either overcorrect on falling or not apply enough corrective control early on in the fall state. Challenges, error sources, and next steps are discussed below.

Project Challenges

The two major project challenges were 1) using a PIC18F2553, and 2) IMU communication struggles.

1) The PIC18F2553 is a great platform, however there were certain nuances that took time through trial and error that didn't arise in class with the PIC18F87K22 used throughout the semester. Major "gotchas" that had to be overcome were figuring out how to connect the PICKIT to program, enabling configuration bits that allowed for in circuit debugging, and having to set certain pins connected to the programming lines as outputs due to environmental grounding issues that would cause erratic behavior when set as inputs. Additionally, the PIC18F2553 being a 5V board and the selected IMU being 3.3V required voltage regulators and additional care to be taken (at least one IMU was burned out during hardware development, potentially as many as three were lost in total).

2) The most complex and frustrating part of this project was communication with the IMU via SPI communication. After several unsuccessful attempts at using the PIC18F2553's built in MSSP unit, a "bit banged" approach was implemented and used software and timing delays to change logic levels on GPIO pins to emulate a SPI communication waveform. Ultimately there are two possible error vectors that have been identified that contributing to an unsuccessful implementation of SPI communication via the MSSP:

    a. Faulty soldering lines: the IMU is mounted on several female headers soldered into the board. Some of the soldered connections of the female headers may not have been totally connected which didn't allow for the IMU response over the MISO jumper cable to reach the PIC.
    b. The PIC18F2553 family of MPUs has several documented instances of the MSSP not working on the Microchip forums. Additionally the datasheet errata identify several issues with the MSSP.

Suspected Error Sources

Three major error sources have been identified in Tipsy's inability to achieve and maintain perfect balance:

1) Cheap Gearmotors: In efforts to reduce cost, cheap gearmotors were used with relatively low max input voltages, output torques and RPM's. Several of the online sources reviewed during the initial feasibility research portion of the project indicated that higher quality motors are imperative in reducing "jitter" and applying corrective control without "overshooting". Although the PWM wave generated from the PID controller could likely be fine tuned, the resolution of the gearmotor may not be responsive to a higher resolution duty cycle.

2) Empirically Tuned PID: To reduce project complexity, an empirically tuned PID controller was used. This made the project more convenient, however without an actual transfer function and knowledge of vehicle dynamics, mass, dimensions, etc, it is difficult to implement a "perfect controller". The empirical tuning was purely through trial an error – K parameters were tuned, uploaded to Tipsy, then

changes in control were observed. The implementation was simple but largely inaccurate "shots in the dark".

3) Sensor Noise/State Estimation: For project simplicity, a simple complementary filter was used. It was able to reasonably estimate poise, however there was a lot of noise from the accelerometer. An attempt to filter this out was made by having a really large α factor, however the noise in the state estimation may have caused unnecessary control input.

Next Steps

This project was very challenging, educational, and inspiring. The biggest takeaways were how to dig deep into datasheets, debug in various methods (logic analyzers, in circuit debuggers, etc), and implement creative solutions when convention isn't working. Future "next steps" to achieve and maintain stable balance include:

1) Larger/More Robust Motors: As noted in the error sources discussion section, better motors with more torque/rpms/duty cycle resolution would likely contribute to less jitter and less unnecessary corrective control.

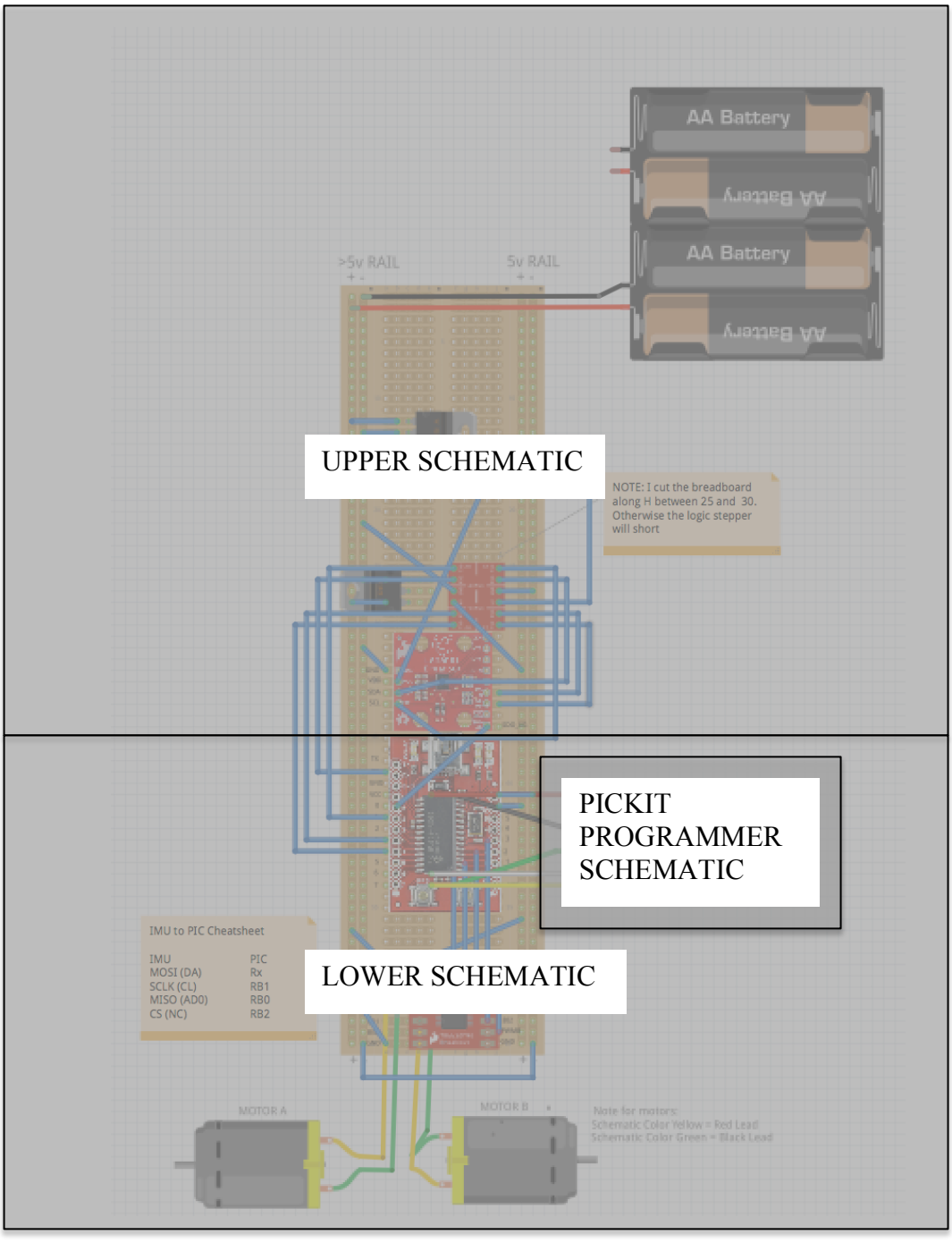
2) Hardware PID Tuning: Using potentiometers and the PIC18F2553's ADC module would likely increase the iterative speed at which the PID K coefficients could be tuned. The version of Tipsy submitted for the final project has the potentiometers built onto the chassis but not wired and no ADC code has been written.

3) Kalman Filtering: A more robust state estimator could help reduce noise and errant control from the noise. Because the PID controller is only basing its control on theta, the control input matrix would be a scalar. If the dynamical model of an inverted pendulum could be simplified into an observable system using just theta and/or theta dot (which is functionally what the complementary filter has done but in a scalar form), then the implementation of a Kalman filter on an 8-bit microcontroller becomes more feasible.

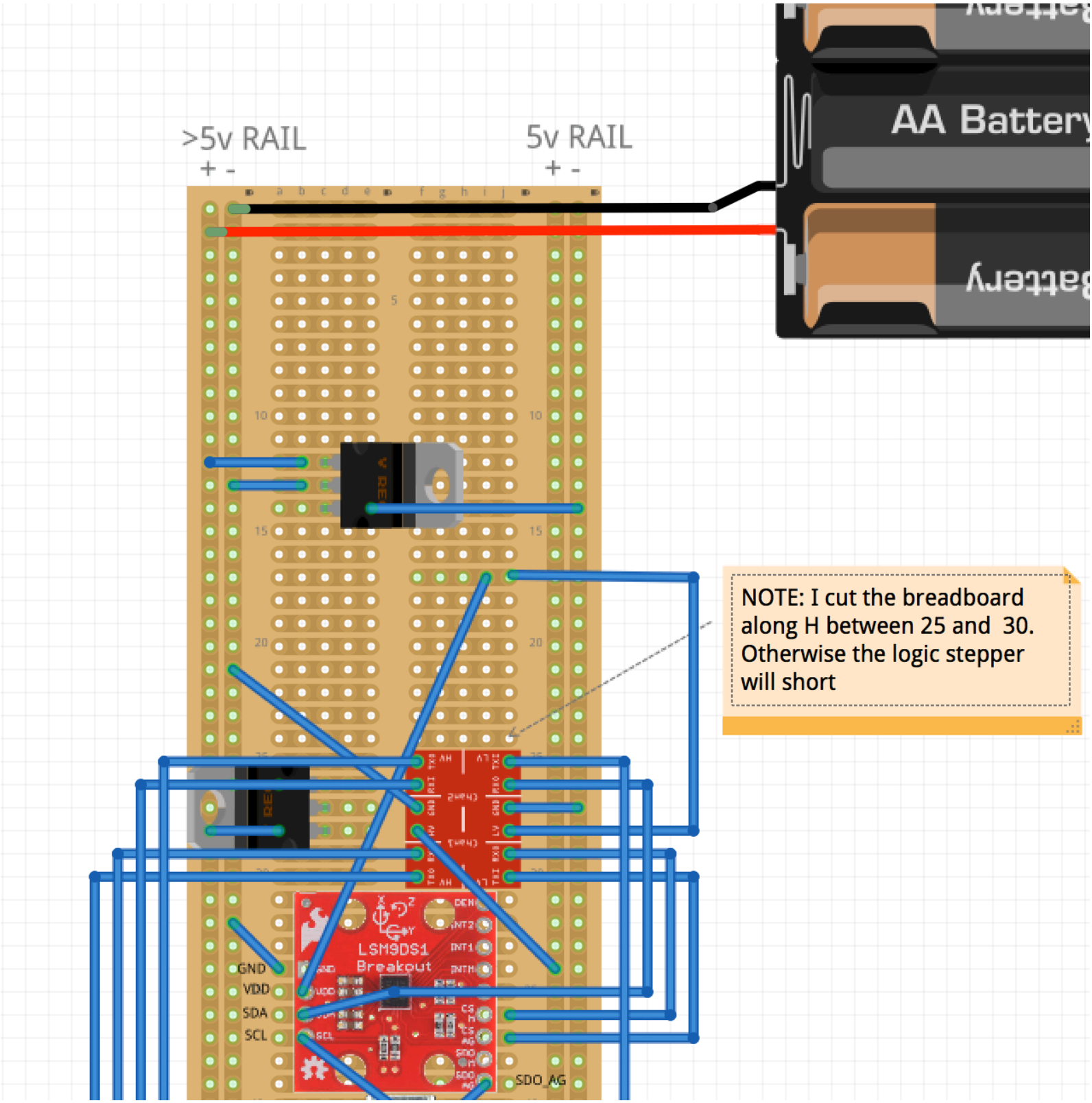
APPENDIX A - WIRING SCHEMATIC

Below is a schematic of the wiring for Tipsy. Power is provided by 4xAA batteries which provide ~6V. Because all components are 5V or 3.3V, voltage regulation was required. Power is provided to a ">5V" rail, then stepped down using 5V and 3.3V voltage regulators. There is a dedicated "5V" rail since the majority of the components are 5V; the 3.3V components are wired individually. A logic level converted is used to step the 5V logic levels from the PIC down to the 3.3V required by the IMU.

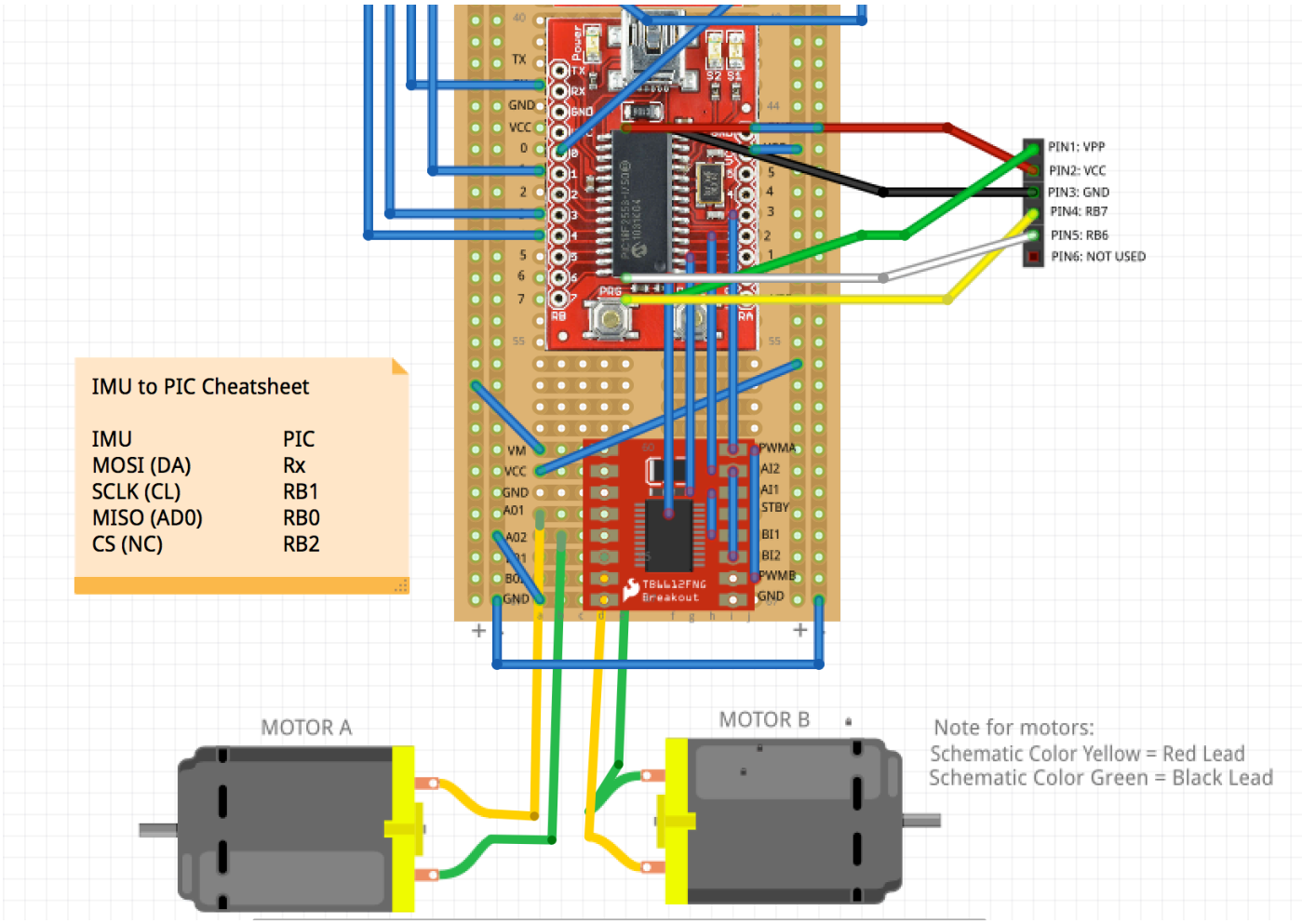
Note that the wiring schematic only details parts that are used in the design. Due to several hardware difficulties, two IMU's have been abandoned in place on the board and do not contribute any functionality to the project.



UPPER SCHEMATIC



LOWER SCHEMATIC



PICKIT PROGRAMMING SCHEMATIC

PICKIT PIN #	PIN DESCRIPTION	PIC18F2553 PIN
1	Vpp/MCLR	Vpp
2	VDD/Target	Vcc
3	Vss (Ground)	GND
4	ICSPDAT/PGD	RB7
5	ICSPCLK/PGC	RB6
6	LVP	Not Used

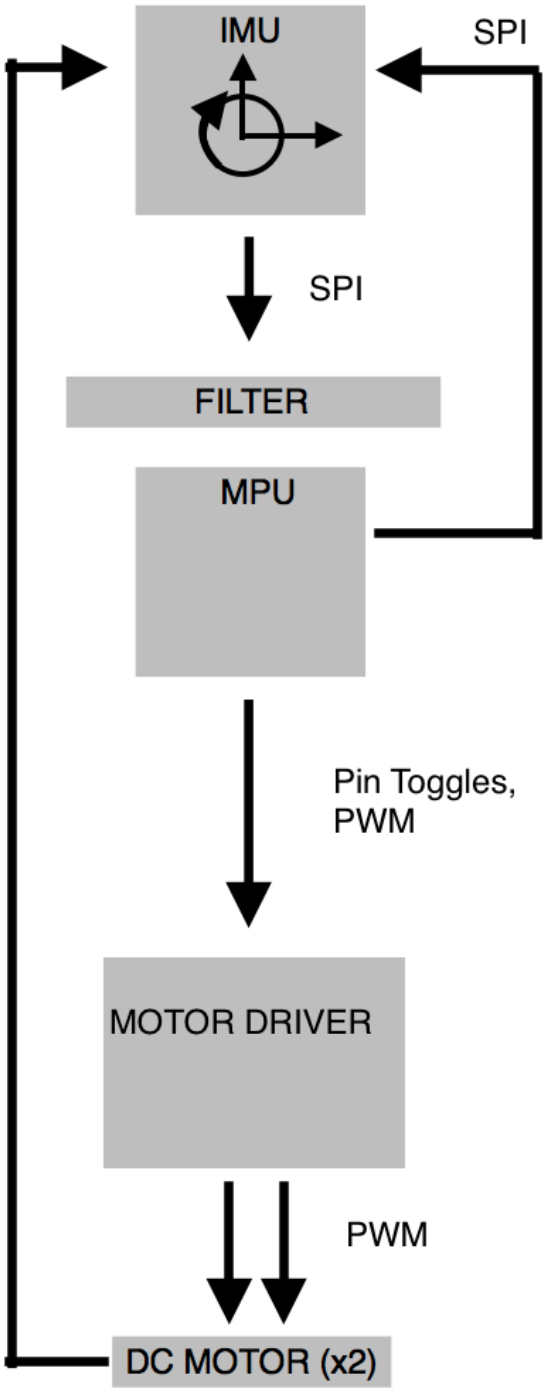
APPENDIX B - BILL OF MATERIALS

Below is a list of the bill of materials used in the project. Note that the bill of materials only include materials that contributed to the final project design – it does not include the several IMU's that were broken.

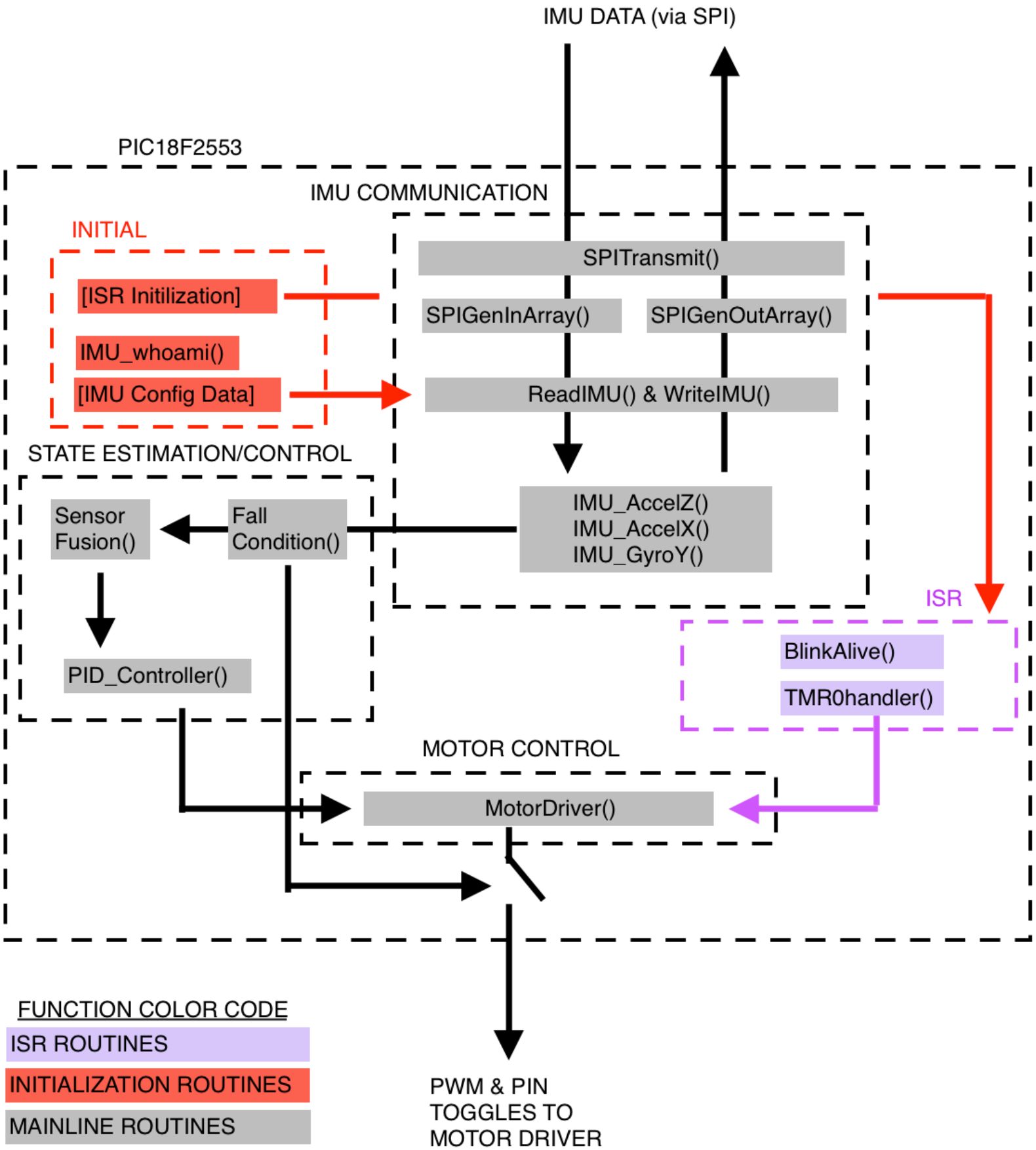
ITEM NAME	UNIT COST	DESCRIPTION
PIC18F2553 Dev Board	$19.95	Development board with PIC18F2553 built in
5V/3.3V Voltage Regulators	$1.90	Voltage regulators for 5V/3.3V perephrials
Battery Holder	$1.95	Holds the batteries
Break away headers, jumpers, screw terminals, etc	~$10	Miscellaneous components used on the board
x2 DC motors	$4.95	Gear motors used for balancing
Chassis Kit	$14.95	Chassis to hold the motors and attach frame
Frame/hardware	~$15	Miscellaneous aluminum angle/screws/nuts/hardware to support the board and perephrials
TB6612FNG Motor Driver	$5.45	Motor driver to drive DC motors
LSM9DS1 IMU Breakout	$15.95	IMU for state estimation
Logic Level Converter	$2.36	Converts between 5V and 3.3V
Breadboard	~$5	Breadboard to hold everything

All in, the approximate project cost of implemented items is approximately $100 factoring in batteries and other miscellaneous items.

APPENDIX C – FUNTIONAL HARDWARE BLOCK DIAGRAM

BLOCK	FUNCTION DESCRIPTION
	The IMU responds to inertial shifts along the X, Z and My axes. MPU reads from IMU via SPI and the IMU sends the signal to the MPU via SPI A complementary filter is used for sensor fusion and to account for accelerometer noise and gyroscope drift to estimate pose Pose estimation from IMU/filter is used to calculate error from vertical. Error is used in empirically tuned PID controller to generate signed motor control scalar. Motor control is interpreted by MotorDriver suborutines to generate proportional PWM signal and toggle motor driver chip pins based on required motor drive orientation (CW or CCW) The motor drivers interpret the PWM signal and distribute it to the onboard DC motors based on toggled pins. A single PWM signal is used and sent to both motors. The actuation of the motors changes the dynamics of the robot and impacts the IMU reading.

APPENDIX D – SOFTWARE FLOWCHART



APPENDIX E – CODE DOCUMENTATION

Code can be found here: https://github.com/rmcsqrd/Tipsy.bot

Subroutine Name	Functionality
Initial() TipsyDriver.c	INPUT: None RETURNS: None DESCRIPTION: Initialization routine that sets appropriate pins to inputs/outputs and sets up TMR0 and Interrupt Service Routine (ISR) for TMR0. Also initializes IMU by writing configuration instructions to appropriate IMU registers via SPI.
IMU_whoami() TipsyDriver.c	INPUT: None RETURNS: None DESCRIPTION: Container function that reads from the "WHO_AM_I" register (0x0F) on the IMU. Used for debugging to validate SPI communication is working.
IMU_AccelZ() TipsyDriver.c	INPUT: None RETURNS: None (updates global variable AccelZ) DESCRIPTION: Container function that reads from high and low Z axis accelerometer IMU registers (0x2D, 0x2C) via SPI. Updates int16_t AccelZ global variable.
IMU_AccelX() TipsyDriver.c	INPUT: None RETURNS: None (updates global variable AccelX) DESCRIPTION: Container function that reads from high and low X axis accelerometer IMU registers (0x29, 0x28) via SPI. Updates int16_t AccelX global variable.
IMU_GyroY() TipsyDriver.c	INPUT: None RETURNS: None (updates global variable GyroY) DESCRIPTION: Container function that reads from high and low Y axis gyroscrope IMU registers (0x1B, 0x1A) via SPI. Updates int16_t GyroY global variable.
TipsyController() TipsyDriver.c	INPUT: None RETURNS: None (updates global variable motor_speed, motor_orientation) DESCRIPTION: Mainline controller function that accesses sensor fusion and control subroutines then updates motor input accordingly. Disables interrupts during functionality to allow for uninterrupted control. Updates global variables uint8_t motor_speed (0-255) and uint8_t motor_orientation (0 or 1).

Subroutine Name	Functionality
SensorFusion() TipsyDriver.c	INPUT: None RETURNS: None DESCRIPTION: Uses a complementary filter (reference "Sensor Fusion of IMU Data") to estimate pose based on global variables AccelZ and AccelX. Updates global variable double theta.
FallCondition() TipsyDriver.c	INPUT: None RETURNS: None (updates global variable fall_status) DESCRIPTION: Parameter check that conditionally evaluates if Tipsy's orientation is within acceptable vertical bounds based on AccelZ variable. If Tipsy has fallen (orientation is > ~45degrees from vertical) the global variable uint8_t fall_status is toggled which overrides motor control and turns on the red indicator LED.
PID_Controller() TipsyDriver.c	INPUT: None RETURNS: None (updates global variable PID_out) DESCRIPTION: PID controller that implements PID control (Dwyer 2014) based on empirically tuned Ki, Kd, Kp variables. Reference report section "PID Control" for theory. Updates global variables double errork, double errork, double PID_out.
BlinkAlive() TipsyDriver.c	INPUT: None RETURNS: None DESCRIPTION: Blinks yellow status LED based on TMR0 ISR.
LoPriISR() TipsyDriver.c	INPUT: None RETURNS: None DESCRIPTION: Handles TMR0 ISR.
TMR0handler() TipsyDriver.c	INPUT: None RETURNS: None DESCRIPTION: Generates the PWM wave that drives the DC motors based on TMR0 ISR timing.
MotorDriver(speed, orientation) MotorDriver.c	INPUT: speed, orientation RETURNS: None DESCRIPTION: Drives appropriate pins to motor driver chip high or low based on orientation variable. Also checks for fall_status value which overrides motor control.

Subroutine Name	Functionality
SPIGenOutArray(data) SPI_Driver.c	INPUT: data RETURNS: None DESCRIPTION: Accepts single byte data and unpacks into global variable char SPI_OutArray[] where each of the array indices matches the bits in data (MSB first, LSB last).
SPIGenInArray() SPI_Driver.c	INPUT: None RETURNS: result DESCRIPTION: Packs values from global variable char SPI_InArray into a single byte of data uint8_t result.
SPITransmit() SPI_Driver.c	INPUT: None RETURNS: None DESCRIPTION: Transmits "bits" (the values at the array indices) in SPI_OutArray[] via a software implementation of SPI communication using software toggles of pins to emulate SCK, MISO, MOSI pins. Reads returned data and updates global variable char SPI_InArray[].
WriteIMU(address, command, reg) SPI_Driver.c	INPUT: address, command, reg RETURNS: results DESCRIPTION: Transmits address and command via SPI to IMU. Accesses appropriate IMU bank register for accelerometer/gyroscope or magnetometer based on reg variable. Implicitly sends a write command because it does not mask a 1 to the MSB (assumes user is attempting write to correct IMU register). Reference report section "IMU Communication via SPI" for more information. Disables interrupts to avoid impacting SPI communication. Returns results variables by accessing SPIGenInArray() to pack up response.
ReadIMU(address, reg) SPI_Driver.c	INPUT: address, reg RETURNS: results DESCRIPTION: Masks address variable with 0x80 (MSB = 1) to initiate a read operation on IMU. Calls WriteIMU() with command byte = 0x00 (arbitrary dummy data).

APPENDIX F – DATA SHEETS

Below is a list of the sensors used and links to their associated data sheets. A separate .zip file with all of the data sheets has been included in the report submission.

SENSOR/PERIPHERAL: PIC18F2553
FUNCTION:MPU
DATASHEET:

http://ww1.microchip.com/downloads/en/devicedoc/39887b.pdf (supplemental)
https://ww1.microchip.com/downloads/en/devicedoc/39632c.pdf (main)

SENSOR/PERIPHERAL: DG01D Gearmotor
FUNCTION: Gearmotor
DATASHEET: https://cdn.sparkfun.com/datasheets/Robotics/DG01D.jpg

SENSOR/PERIPHERAL: LSM9DS1
FUNCTION: IMU
DATASHEET:
https://cdn.sparkfun.com/assets/learn_tutorials/3/7/3/LSM9DS1_Datasheet.pdf

SENSOR/PERIPHERAL: TB6612FNG
FUNCTION: Motor Driver
DATASHEET: https://www.sparkfun.com/datasheets/Robotics/TB6612FNG.pdf

SENSOR/PERIPHERAL: BSS138
FUNCTION: Logic Converter
DATASHEET:

https://cdn.sparkfun.com/datasheets/BreakoutBoards/BSS138.pdf (datasheet)
https://learn.sparkfun.com/tutorials/bi-directional-logic-level-converter-hookup-guide
(hookup guide)

APPENDIX G - REFERENCE

Below are some of the materials referenced over the course of the project. All code was the original work of the author unless noted otherwise.

"Self Balancing Robot", Brian Dwyer 2014:
http://ohmwardbond.blogspot.com/2014/12/self-balancing-robot.html?m=1

"SparkFunLSM9DS1.cpp", Jim Lindblom 2015:
https://github.com/sparkfun/SparkFun_LSM9DS1_Arduino_Library/blob/master/src/SparkFunLSM9DS1.cpp

"Inverted Pendulum: PID Controller Design", Control Tutorials:
http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=ControlPID