Microcontroller Technical Manual

Samuel Tseng

June 12, 2019

Contents

1 Quick Start Guide

An online version of this document can be found on our GitHub at:
https://github.com/AMDHome/Mechanical-Speedometer-Converter/tree/master/controller
(GitHub > controller > README.md)

1.1 Installation

Below is a wiring diagram of the device we have provided you

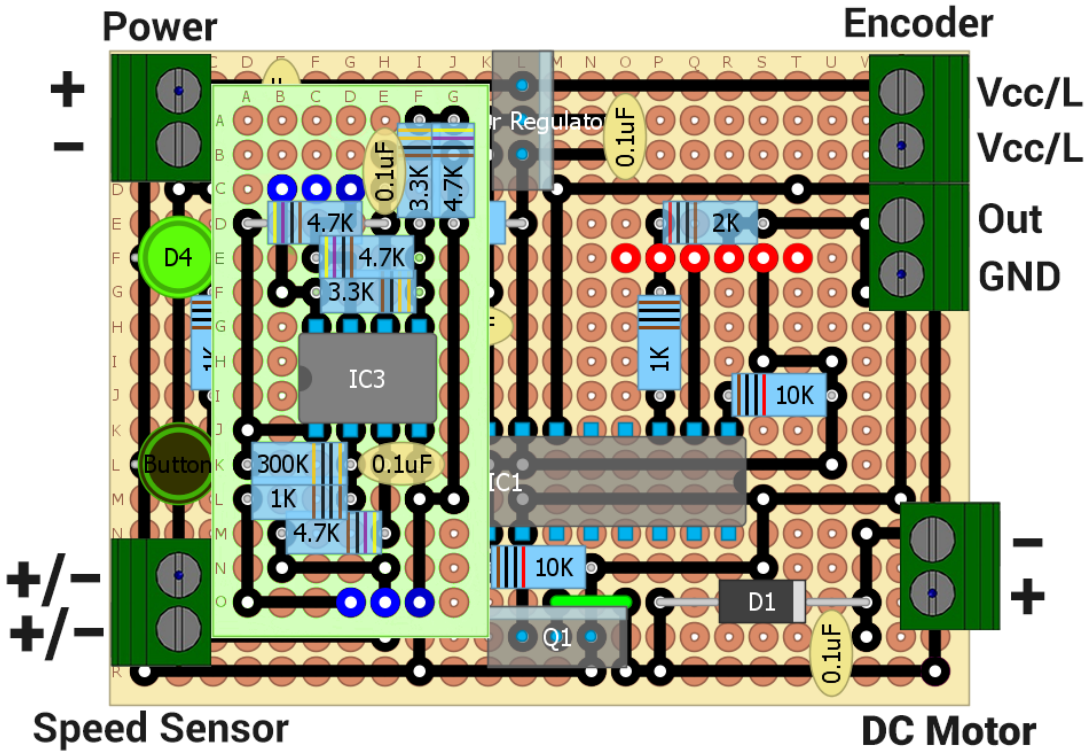


Figure 1: Wiring Diagram

Please note that for the speed sensor provided you can wire it in either direction. It does not affect the operation of the speed sensor. Same with the encoders Vin and L pins, they can be swapped. Everything else must be wired in exactly as shown.

For the DC Motor, we assume that the terminal with the red dot corresponds with the + terminal and should be wired as shown (Positive on bottom).

Please make sure the bottom of the device is on a non-conductive surface as it is just bare traces. Anything conductive will short circuit the device and potentially burn something.

1.2 Setup

After installing the app and connecting the device as shown, power on the device (via your car or an external power supply), open the Bluetooth settings on your phone and pair the phone to the Bluetooth module. It should be named DSD HC-05. When prompted for a password enter 1234.

Once paired you can launch the app and click on devices.

From there you can select the microcontroller. Once selected it should automatically load values from the microcontroller and you should be able to set the values to whatever you wish.

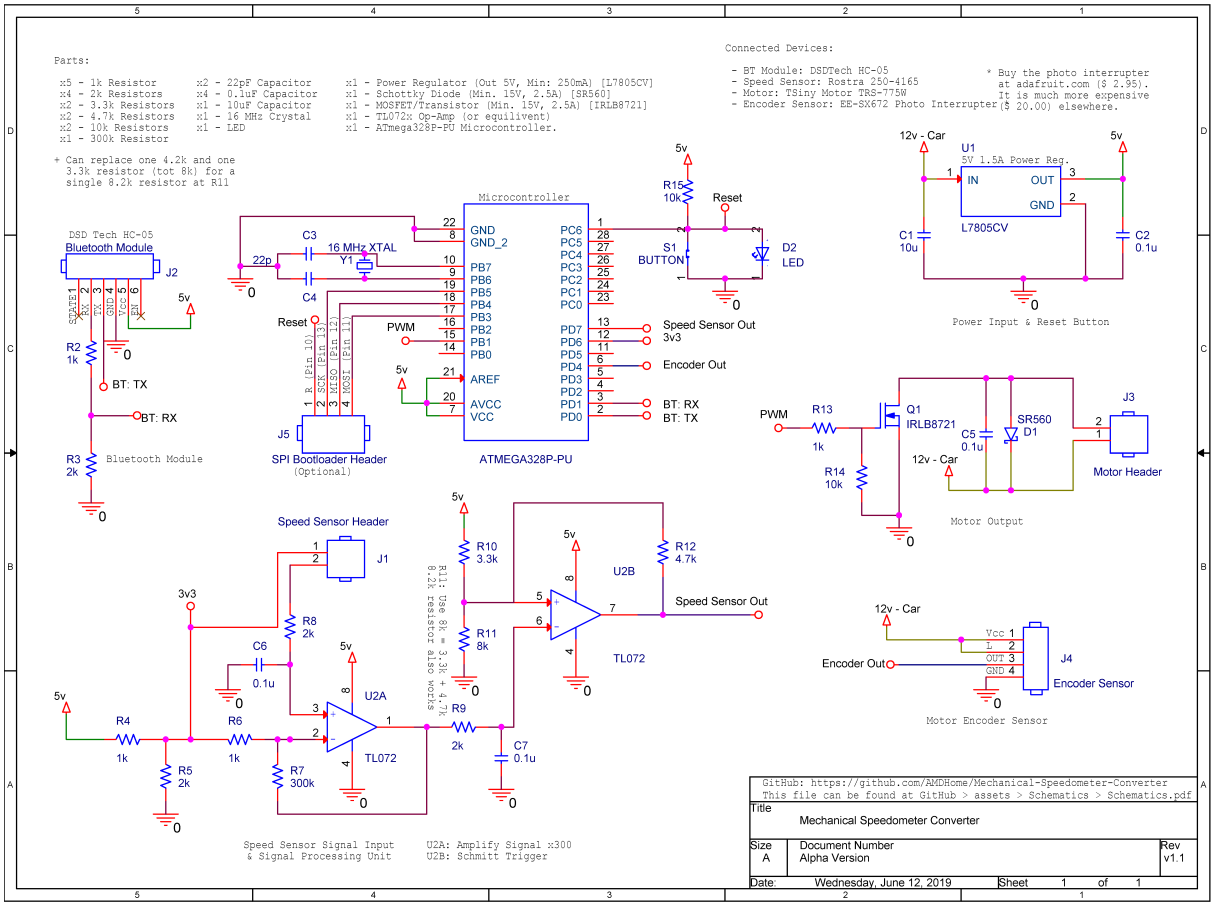
1.3 Troubleshooting

1. If the app crashes or says the values failed to load, it means we need to do a hard reset. To do this please do one of the two things
    - Resetting via Bluetooth Serial Terminal App
        - Load up a Bluetooth serial terminal app, connect it to the microcontroller, and type `L:1`
        - Check to make sure the values are all numerical values. If any characters are not numbers then that is the problematic value
            * Please note the wheel size value will have non numerical values. It must be in the format `P___/__R__`
        - Cross reference the output with this line to figure out what value is causing the issue: `L:Units:Max_Spd:Mags:Final_Drive:Speedo_Ratio:Wheel_Size_Text`
        - Once you know what value is causing the issue, go to the General Commands Format table to find the command corresponding to the problematic value
        - type X:0 where X is the problematic value's command letter
        - Restart the microcontroller by either power cycling it or push the button found on the device. The microcontroller will automatically reset that value to the default value
    - Resetting via Arduino programmer
        - Remove the ATMega328P chip and plug it into your programmer/Arduino.
            * I typically use the Arduino Uno to do all my programming for me
        - Run the reset.ino script found on the GitHub
        - wait at least 5 seconds after it says upload complete
        - Run the controller.ino script found on the GitHub
        - remove the ATMega328P chip from the programmer and put it back into your microcontroller.

2. If the microcontroller appears to be frozen or broken try resetting it. It might have accidentally been put into an undefined state and gotten confused
3. If the app says it cannot send values, kill the app. Make sure the microcontroller's Bluetooth LED is blinking fast. If it isn't, restart the microcontroller. You can now reopen the app and reconnect. It should be fixed now.

2 Hardware

Below is a schematic of our microcontroller. This section will go through it in detail as to what everything does.



This a larger version of this image can be found on our GitHub at:
GitHub > assets > Schematics > Schematics.pdf

2.1 ATMega328P Pinout

Our microcontroller uses the ATMega328P as its SoC/CPU. Here is the pin layout that we use.

Pin # / Name	Feature	I/O	Connection
1 (PC6)	RESET	Input	Reset Button
2 (PD0)	Rx	Input	Bluetooth Input
3 (PD1)	Tx	Output	Bluetooth Output
4 (PD2)	-	-	Not Used
5 (PD3)	-	-	Not Used
6 (PD4)	T0 Input	Input	Encoder Input -> Timer0
7	Vcc	Input	Power In (5v)
8	GND	-	Ground
9 (PB6)	XTAL1	Input	Clock Pin 1
10 (PB7)	XTAL2	Input	Clock Pin 2
11 (PD5)	-	-	Not Used
12 (PD6)	AIN0	Input	3v3 Input to Comparator
13 (PD7)	AIN1	Input	Speed Sensor Input to Comparator
14 (PB0)	-	-	Not Used
15 (PB1)	OC1A	-	Motor PWM Output
16 (PB2)	-	-	Not Used
17 (PB3)	MOSI	Input	Used for Burning Bootloader
18 (PB4)	MISO	Input	Used for Burning Bootloader
19 (PB5)	SCK	Input	Used for Burning Bootloader
20	AVcc	Input	Redundant Power Input (5v)
21	ARef	Input	Reference Voltage (5v)
22	GND	-	Ground
23 (PC0)	-	-	Not Used
24 (PC1)	-	-	Not Used
25 (PC2)	-	-	Not Used
26 (PC3)	-	-	Not Used
27 (PC4)	-	-	Not Used
28 (PC5)	-	-	Not Used

2.1.1 Reset Pin 1 (PC6)

This pin needs to remain high as long as the microcontroller is to be running. Pulling this to ground will cause the microcontroller to power off, even if Vcc is connected. The button here will pull the pin to ground and act as a reset switch. The LED is there to show when the microcontroller is operational.

2.1.2 XTAL Pins 9 & 10 (PB6/7)

These pins connect to a 16 MHz crystal. This acts as the system timekeeper as all the timings are some multiple of 16 MHz. Any alterations to this value with cause the system timings to change. Each leg of the crystal is connected to one 22 pF bypass capacitor to help remove spikes from the input.

2.1.3 SPI Pins 17, 18 & 19 (PB2/3/4) [Optional]

These pins are for an external header to help you program the bootloader of the ATMega328P chip when you are unable to remove it (surface mount). If you are able to remove the chip, you do not need to have this header to program the microcontroller, instead you can just remove the chip and plug it into a breadboard/Arduino. These pins are not wired to the microcontroller we gave you.

2.2 Power Input

This part is fairly straightforward. We use a power regulator to convert 12v power to a steady 5v for our system. The two capacitors `C1` and `C2` are bypass capacitors to help smooth out the power.

2.3 Bluetooth Module

This one is also a very simple circuit. We provide power to the Vcc (5v) and ground pins and then map the serial pins from the Arduino to the Bluetooth. We have a voltage divider to the RX pin of the Bluetooth as the Bluetooth module's logic circuit is only 3.3 volts.

Please note that the Rx pin on the Bluetooth module should be connected to the Tx pin on the ATMega328P and vice versa. This is because the module is receiving whatever the ATMega328P is sending. The same is true for the other pairing

2.4 DC Motor Output (PWM)

This is a pretty standard PWM circuit that drives the motor which will allow us to spin the speedometer. Our PWM passes through a MOSFET which will provide 12v power based on the PWM signal. We have a Schottky Diode as well as a small bypass capacitor to help smooth out any ripples that are generated by the motor.

2.5 Encoder Sensor

This really isn't a circuit; we just provide power to the sensor and it takes care of everything. The only thing is the L pin which needs to be connected to 12 volts and it puts the encoder into the proper state.

2.6 Speed Sensor ADC

This is by far the most complicated circuit that we have designed. We start off by running the signal through a low pass filter (R8 & C6) to limit the speed sensor input frequency to 796 Hz. This will help get rid of some noise and 796 Hz should be more than enough for our purposes (180 MPH with a configuration of 4 magnets on the drive shaft, final drive of 5, and wheel circumference of 80 in).

After running it through the low pass filter we then put it through an Op-Amp (U2A) to amplify the signal by ~301 times. This normalizes the signal so that no matter the amplitude of the input signal, the resulting signal will be between 4.5v and 0.5v. This is required as when driving at slow speeds as our signal only has a peak-to-peak voltage of 15 mV.

Afterwards, we run it through another low pass filter (R9 & C7) to filter out the noise generated by the amplifier. This probably isn't necessary. It's just here because I got fed up noise when designing the circuit and leaving it in produced a more stable signal for me to work with; however, I **think** the circuit is robust enough now that if you removed it, it would be fine.

Finally, we run the signal through a Schmidt trigger that is built with the second Op-Amp on the chip. A Schmidt trigger is a device that will create a digital signal based on an analog signal. It has two thresholds, a high and a low. The analog signal must pass the high threshold in order for the output to be high and it must cross the low threshold for the signal to turn back to low. This allows the device to counteract some noise in the system. For an example please see the diagram on the next page. The green lines and waveform are the result of a Schmidt trigger acting on wave U.

For our Schmidt trigger, our thresholds are around 4v and 1.1v. Since our Schmidt trigger is built using an op-amp and not a comparator, the output signal is only pseudo-digital. To fix this we just run the resulting output through the analog comparator built into our ATMega328P processor. These four steps produce the necessary digital output for us to read the speed sensor.

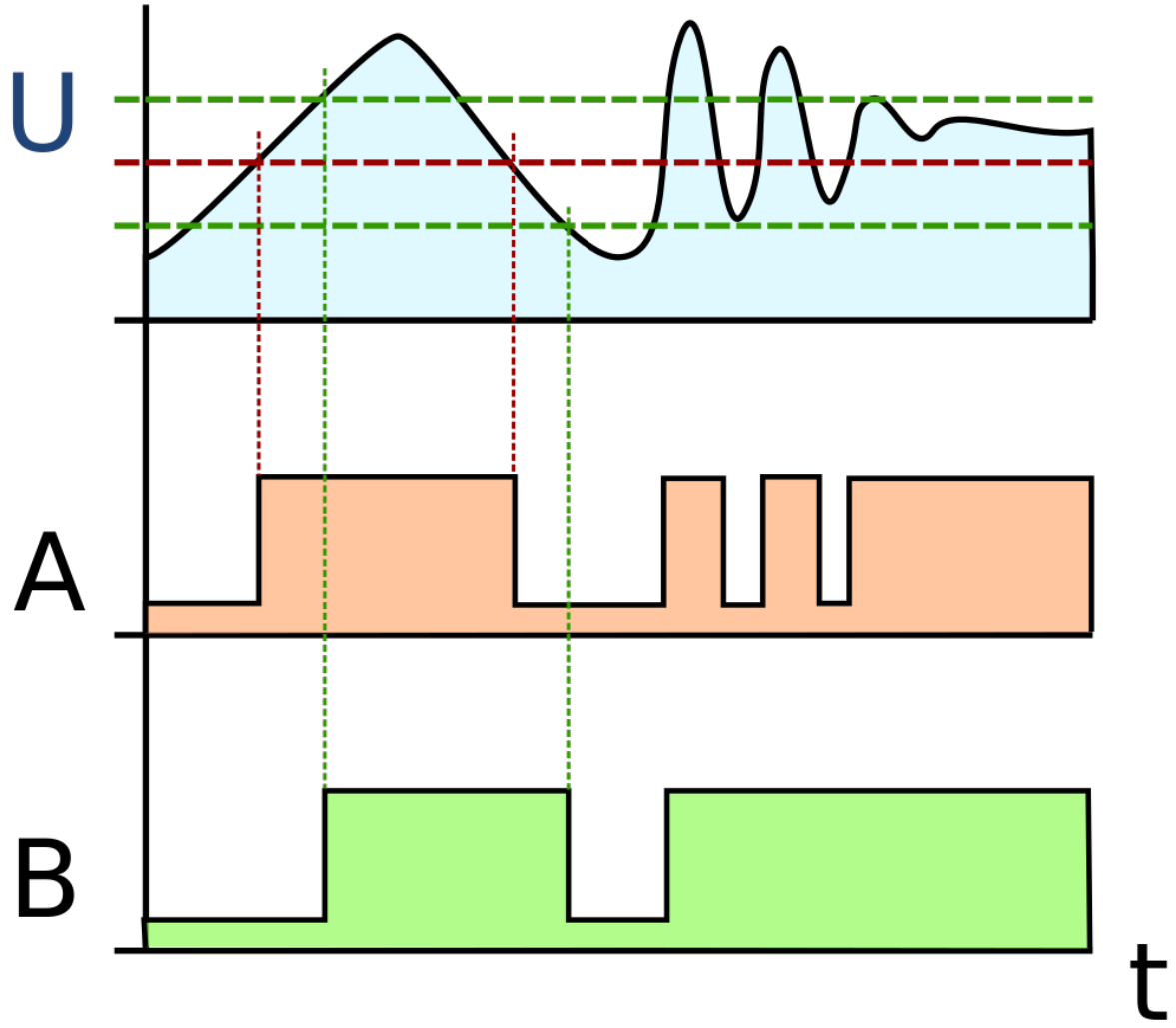


Figure 2: Schmidt Trigger Example (Wikipedia)

You may have also seen that we run a 3v3 power through the speed sensor, this is to raise the entire wave up so that it works properly with our op-amps. The op-amps do not like processing a signal near their thresholds (for us the op-amp thresholds are GND and 5v).

3 Programming the ATMega328P

3.1 Burning the bootloader

When you buy a brand new ATMega328P chip, chances are it doesn't have the Arduino bootloader on it. In order to program the chip, we need to first burn a bootloader to the chip. Luckily, we can use an Arduino to do this fairly easily by following the instructions at the following link: **https://www.arduino.cc/en/Tutorial/ArduinoToBreadboard**

If you wish to use the surface mounted version of our chip, or a version that cannot be removed, you will need to use the optional SPI headers that we have diagrammed onto our schematic. You will still follow the guide in the link above. If doing it via the built-in headers you will need to make sure the wires are connected to the correct pins. Pins 10, 11, 12, 13 will be connected to the Optional SPI Bootloader header while the 5v and GND pins should be connected to the Bluetooth's Vcc and GND pins respectively. This will allow you to burn the bootloader without removing the chip. The proper pin mappings for the header are labeled on the schematic. You are welcome to reorder the pins as you see fit.

Once again, the optional bootloader header does not exist on the microcontroller we gave you.

3.2 Uploading the program

Once you have your bootloader installed, we need to upload the program. You can do this either by following the procedures in the link above, or by plugging the chip directly into your Arduino and doing it from there (only works with the removable 28-dip chip)

If you are not able to remove your chip you can wire the Arduino to our microcontroller exactly how it shows in the picture. Tx will connect to the BT:Rx header and Rx will connect to the BT:Tx header (Yes, they are flipped, See the BT Module section above). 5v and GND can be connected to the Bluetooth's Vcc, GND pins and reset will be connected to the Optional SPI Headers reset pin.

Once you have everything wired up, we first need to upload reset.ino. This will store all default values into the ATMega328P's permanent storage.

Once that is complete, we can upload controller.ino. This will provide it with the program that it needs to run everything.

4 Software

Here we will talk about the code in great depth.

4.1 Microprocessor (ATMega328P)

Before we begin, we need to talk about the hardware features that the microprocessor has built in. We will only mention the parts relevant to the codebase.

4.1.1 Clock

Our Microprocessor has a 16 MHz clock connected to it (produce a signal with 16,000,000 cycles per second)

4.1.2 Timer/Counter

A Timer/Counter is a register that counts up automatically when some sort of signal is provided to it. If the signal provided is of a consistent frequency it becomes a timer that counts up at set intervals. These counters do not take up time on the main CPU so they can run separately without disturbing the main code.

Each one of these Timer/Counters consists of the main register that stores the count as well as a prescaler [1] and some auxiliary registers to program the counter.

[1]: A prescaler is a device that scales back the frequency of a signal.
`Input_Frequency / Prescaler = Output_Frequency`

From this point forward we will use the words Timer and Counter interchangeably to reference these devices

On our processor we have three timers (T0, T1, T2). Below is a flowchart of the processes that drive these three timers. You can find a larger version of the flowchart in the online version of this document.

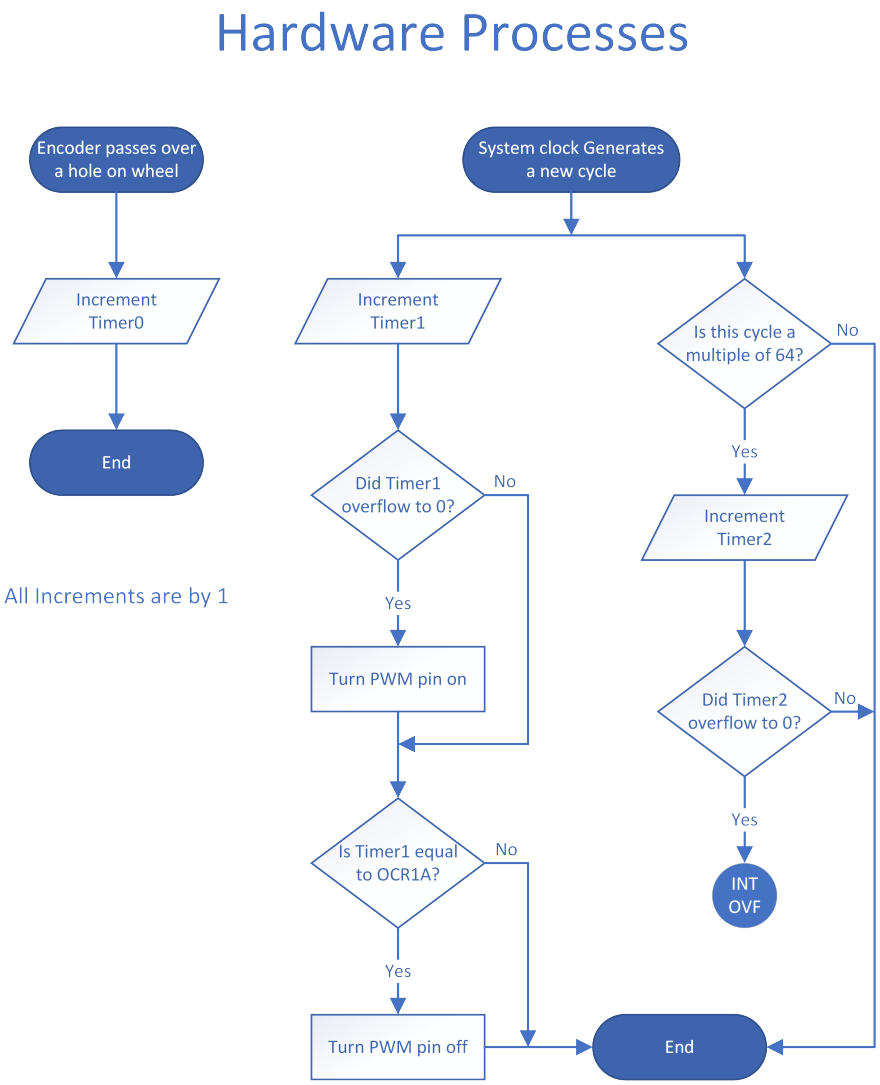


Figure 3: Hardware Processes Flowchart

T0 This is an 8-bit timer that is used to count the number of holes that the motor encoder passes by. Every time it sees a hole the value in T0 gets incremented by 1.

T1 This is a 16-bit timer that is configured to operate in 10-bit mode.

This timer has a prescaler of 1 so every time the clock cycles the counter will increment by 1. Since it is operating in 10-bit mode it will reset to 0 after reaching 1023 (`2^10 - 1`).

We use this timer to produce our PWM signal. We do this by setting a number between 0 and 1024 in one of the auxiliary registers (`OCR1A`) and every time the counter is incremented, it will check with the auxiliary register. If the values match then it will turn off Pin 15 on the microcontroller. The pins automatically turn on when the counter overflows to 0.

T2 This is an 8-bit timer that is used as our system clock. It has a prescaler of 64 (increments by 1 every 64 clock cycles). This is the clock that we use when we need to calculate the amount of time that has passed.

Each count in this timer is equal to 4 microseconds. You can obtain this number by dividing the prescaler by the clock frequency `(64 / 16,000,000 = 0.000004)`.

Since this is an 8-bit timer, the timer can count up to 255 `(2^8 - 1)`. This means that the total amount of time that the timer can count is 1.024 milliseconds `(4 µs * 256 = 1024 µs)`. To combat this short amount of time. Every time the counter counts to 1024 µs, an interrupt is called to record the overflow. This allows us to keep track of longer amounts of time. The overflow function `ISR(TIMER2_OVF_vect)`; can be found at `wiring2.cpp:36`. This function also helps keep track of other things that we need to time, which will be discussed later in the interrupt section

4.1.3 Analog Comparator

A comparator is a device that takes in two voltages and outputs a digital signal `(HIGH/LOW)`. Our processor has a comparator built into it on pins 12 and 13. We use this comparator because our speed sensor's Schmitt trigger was built using a spare op-amp instead of a comparator. This means that the Schmitt trigger output is an analog square wave that has some noise. We then just use the built-in comparator to convert the signal to digital by checking to see if it goes above/below 3.3v.

4.2 Code Layout

All of our code is put into 5 files. Below are the different parts of code that are in each file:

- `controller.ino`: Setup and Main Loop
- `BTComms.h/cpp`: All Bluetooth I/O is taken care by these two files
- `wiring2.h/cpp`: All timing functions are in this file. It is a modified version of Arduino's original `wiring.cpp` and timing functions

4.3 Main Code

4.3.1 Overview

This is the most straightforward part of the code. When the device starts up it runs `Setup()` once and then it runs `Loop()` over and over again.

4.3.2 `controller.ino:Setup();`

Here we set up the state of the microcontroller. In total it does 3 things:

- Set pinmodes for manually controlled pins
- Load in stored specs for your vehicle
- program the hardware components (timers and the analog comparator).

Information for programming the hardware components can be found in the ATMega328P Manual at the following link: **https://www.sparkfun.com/datasheets/Components/SMD/ATMega328.pdf**.

4.3.3 `controller.ino:Loop();`

Here we run the bulk main processes. Below is a flowchart of the entire loop process. You can find a larger version of the flowchart in the online version of this document.

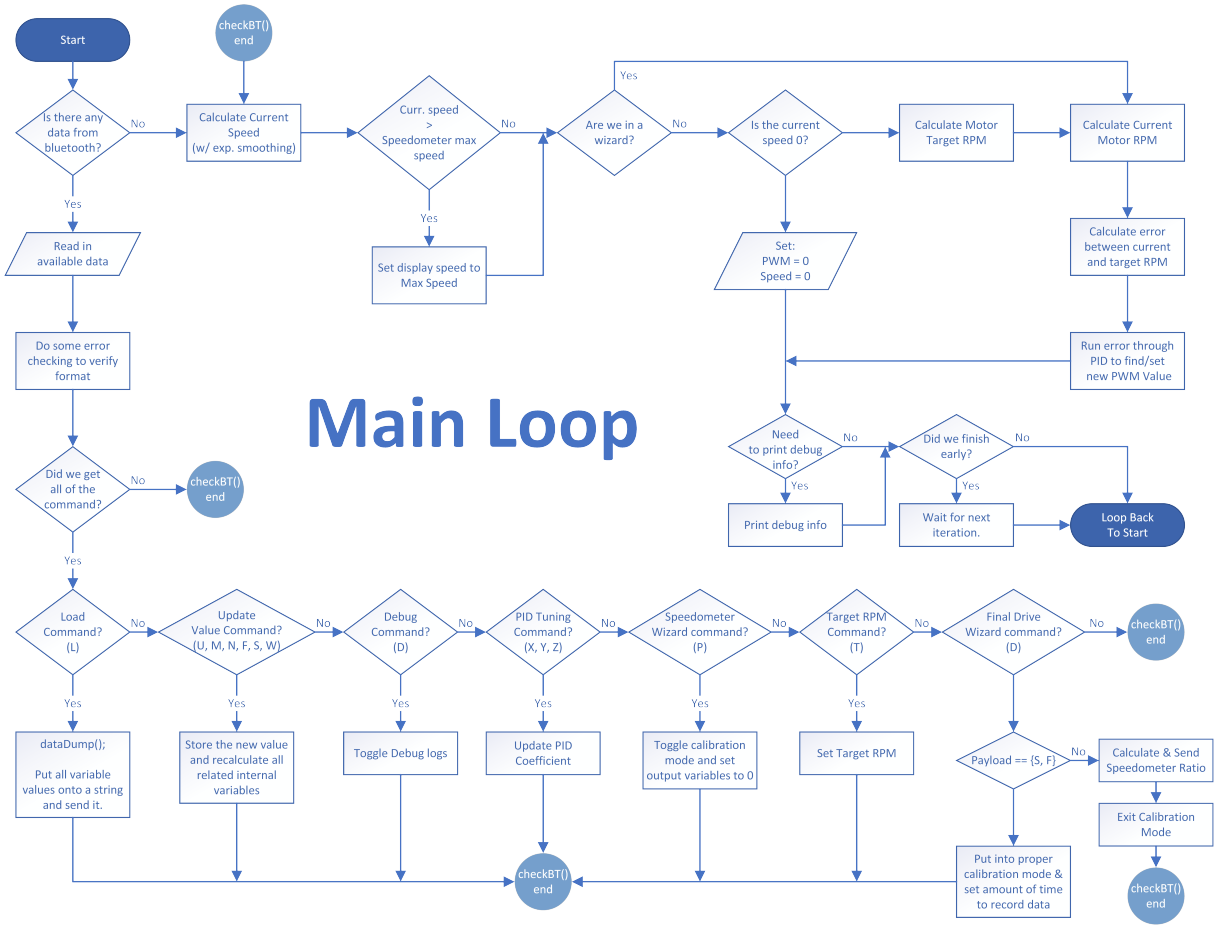


Figure 4: Main Loop

On the next page is a description of the process in text form:

Please Note: The flowchart above includes the Bluetooth processing portion. We have left that out of this section. You may find a description of the Bluetooth processes in the Bluetooth section below.

1. Check Bluetooth to see if any signals come in, if so, go deal with the signal
    - See Bluetooth section below for more details
2. Calculate current speed
    - Explanation of the equation will be in the equations section below.
3. Make sure the current speed can be displayed on the speedometer
    - If the current speed is too high then we just show the maximum speed
4. Calculate the target RPM we want the motor to spin at.
    - If the current speed is 0, we just stop the motor.
    - If the app put us in a special mode (i.e. Wizards), we also stop the motor to prevent undefined behavior
    - If we manually assign the targetRPM (only in Speedometer Wizard) then use the assigned value as the targetRPM
    - If neither of the cases above are true then it is just `current speed * speedometer ratio`. The extra constants are just unit adjustments so we can avoid using floats. More information in the equations section.
5. Calculate the current RPM
    - Explanation of the equation will be in the equations section below.
6. Calculate PID and adjust PWM to control motor
7. Check if it needs to print out the debug logs
8. Stall for time until it needs to start the next iteration
    - This delay is added in because PID works best when each iteration is about the same time period apart
    - This delay can be removed to improve responsiveness; however, it may increase the difficulty of tuning the PID.

4.4 Interrupts

4.4.1 Overview

In our code we have two interrupt loops. We use these loops to run code alongside the main loop. As the name suggests these code blocks have a higher priority as they are time sensitive, and will interrupt the main loop when a certain event is triggered. We will discuss the two loops in more detail on the following page.

In the meantime, here is a flowchart of the interrupts:

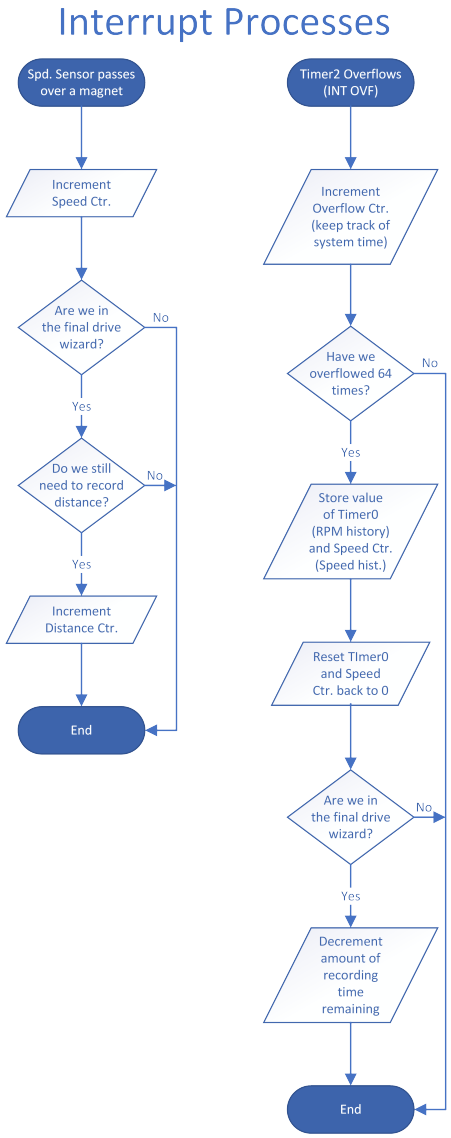


Figure 5: Hardware Processes Flowchart

You can find a larger version of the flowchart in the online version of this document.

4.4.2 `controller.ino:ISR(ANALOG_COMP_vect);`

This interrupt is triggered when a magnet passes over our speed sensor. All it does is increment a counter that tells us how many magnets have passed in this time frame. If we happen to be in the final drive wizard, we will also increment a counter specifically for that wizard to record distance.

4.4.3 `wiring2.ino:ISR(TIMER2_OVF_vect);`

This interrupt is triggered whenever Timer2 overflows (every 1.024 ms) and it does 4 things

1. It increments counters to keep track of longer periods of time
    - this is standard for Arduino code and is usually done elsewhere
2. Every 64 calls of this function we will save the values from the speed sensor counter and Timer0 (encoder data) into two separate circular arrays.
    - This 64 value is kind of like a software prescaler for recording speed data.
    - This will happen once every 65.536ms or 15.26 samples a second.
        - This 15.26 number is important. From now on we will call this the **`sample rate`**
    - Why did we choose this value? it was arbitrary. We felt like collecting data from the sensor around 15 times a second was enough and the numbers played nice with our equations. This number can be changed but the equations will need to be recalculated.
3. Reset the values we just read from to 0 to prepare for the next reading
4. If we are in the final drive wizard reduce the number of samples that we still need to collect

4.5 Wizards

4.5.1 Overview

Because we implemented wizards in the app, we need to set special states for the device to be in. This is because some of the values will be undefined when the wizards are running and might damage the speedometer if we don't put them in these specialized states.

Here is an overview of how the wizards work on the side of the microcontroller.

4.5.2 Speedometer Ratio Wizard

In this wizard we expect the user to be the feedback for the calibration

- On the app side:
  - The speedometer wizard allows the user to adjust the speedometer ratio until the speedometer reads a predefined speed.
  - The app will continuously calculate a target RPM based on the inputted speedometer ratio and the predefined speed and send it to the microcontroller
  - If the speedometer reading is too low, the user needs to increase the speedometer ratio. If it's too fast, they need to decrease it.
- On the microcontroller
  - The Bluetooth will read in a targetRPM from the app
  - It will then skip calculating the targetRPM as to not overwrite the value that was read in.
  - Using the targetRPM that was read in, it will use PID to adjust itself until it spins at that speed.
  - The user will then increase or decrease the value depending on what they need for their speedometer.
  - When the user is happy with the value, the new speedometer will be stored and the microcontroller will be taken out of the Speedometer Calibration mode

4.5.3 Final Drive Wizard

In this wizard we expect the user to drive and hold a constant speed so the microcontroller can sample the sensor's rate at a specific speed. The microcontroller will then take the rate and speed to calculate a final drive.

- On the microcontroller
  - The targetRPM is set to 0 at all times so the speedometer will not run with a potentially faulty final drive value. The main loop will only be used for communicating over Bluetooth. Everything else is handled by the two interrupt loops
  - The Bluetooth will read in the start signal when the user has reached sufficient speed and start recording the number of sensor readings in a period of ~10 seconds `(153 samples = sample_rate * 10)`.
  - If the app restarts the countdown then it will send the start signal again at which the microcontroller will restart this process.
  - After 10 seconds the microcontroller will then wait for an average speed from the user's phone.
  - With the number of readings, it collected and the average speed, the microcontroller can now calculate the final drive. Once calculated it will send the value back to the phone to be displayed on the menu.
  - Equations for the final drive ratio will be below in the equations section

4.6 Bluetooth

4.6.1 Overview

During the main loop, the program will check the Bluetooth module for any new data, we will now go through the Bluetooth section of the main loop here.

Below is the entirety of the main loop again for ease of reference.

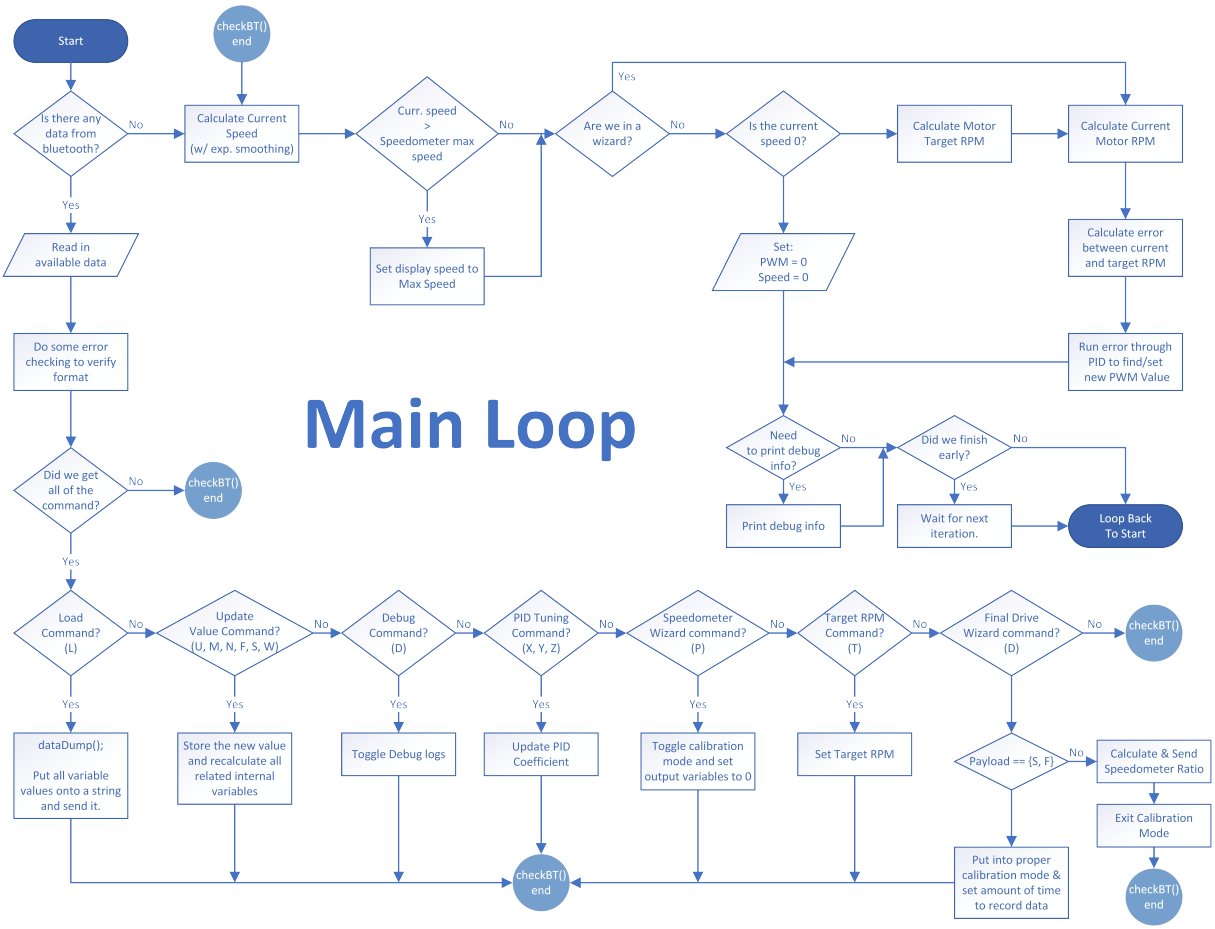


Figure 6: Main Loop

On the next page, we will start talking about the basic formats we use to communicate between the app and the microcontroller.

4.6.2 Basic Communication Format

All commands sent to/from our microcontroller follow a very basic format. Below is an example of a command:

```
S:18000000
```

Format:

1. We start the command off with a capital letter. This is an identifier that tells us how to processes this command.
2. The second character is always a colon. This is here just for human readability for now. In the future, if you run out of identifiers, you can increase the start of the command to 2 letters or more and use the colon as a delimiter.
3. The remainder of the command is the payload. The payload can have anything in it as long as it is less than 23 characters long. This is an arbitrary length that suits our purposes and can be increased in the future if you run out of room.

The program in its current state makes 3 assumptions in the command that it receives:

1. The command follows the format above
2. The commands ends with a null character `'\0'`
3. Capital letters can only be at the start of the command or right after a colon.

Any command added in the future that does not follow these assumptions will need the Bluetooth functions re-coded to support it. One such assumption that has already been coded in is the capital letter `R`. Since this letter appears in the tire size as the radius, we have coded it to not trigger the error checking

4.6.3 Reading the Data In

We do all of our data receiving in the function `BTComms.cpp:recvData();`

This function reads in the available data one byte/character at a time. For each character read in, it does the following checks:

1. Capital letter:
    - If the byte that it just read in is a Capital letter, then it checks to makes sure that it is in a valid location (see assumption number 3 in the previous section).
    - If the capital letter does not meet the assumptions from before then it will assume that the ending of the previously read data was corrupted and that this capital letter is the start of a new command. It will then discard all previously read data and set this capital letter to be the first character read

2. NULL character `'\0'`:
    - If the byte that was just read in is a NULL byte, the program assumes that the command has terminated.
    - It will check the first character read to see if it is a capital letter.
    - If both of these assumptions are met, it will then process the command.
    - If one or more of the assumptions are broken then it will assume the command is corrupted and delete all previously read data
3. Anything else:
    - If neither of those cases occur, then it will just continue to read in data.
        - If there's nothing more to read, it will go back to running the main loop. When there is something else to read, the program will continue from where it left off.

4.6.4 General Commands Format Table

Here is a list of all commands that are currently sent from the app, what the command does, the data type that we use to store the commands on the microcontroller, as well as some expected values.

Cmd	Description	Expected Values[1]	Example Command
U	Set **U**nits	0: KMH, 1: MPH	U:0
M	Set **M**ax Speed	0 - 300	M:120
N	Set **N**umber of Magnets	1 - 4	N:2
F[2]	Set **F**inal Drive Ratio	0 - 10,000,000	F:1000000
S[3]	Set **S**peedometer Ratio	0 - 100,000,000	S:18000000
W[4]	Set **W**heel Size	0 - 10,000,000:MTS[5]	W:1263341:P205/65R15
P	Put/Exit Microcontroller into S**p**eedometer Ratio Wizard	0: Start, 1: Stop	P:1
T[6]	Set **T**arget RPM (For Speedometer Wizard)	0 - 120,000	T:30000
L[7]	**L**oad Values (For App Startup)	1	L:1
D[8]	Start/Stop/Finish Final **D**rive Ratio Wizard	S/F, 0 - 500	D:S, D:400

Wheel Size Command

This is one of our most complicated commands we have implemented. This command's payload is split into two parts

1. The first part of the payload is the circumference of your tires in your chosen units (miles/kilometers) multiplied by a billion. We do this to leverage the phones CPU power and so we don't have to convert it every time ourselves.
2. The second part is just the tire size as regular text. This should be directly off the side of your tire. We use this value to send to the app upon connecting to our device

[1] All values except for final drive are stored with all adjustments. Final drive is the only value that is converted back to the original value.
[2] These numbers are multiplied by a million so we don't have to deal with transferring decimals over commands.
[3] See footnote 2
[4] See the Wheel Size Command subsection on this page for a description of the payload for this command.
[5] MTS -> Metric Tire Size
[6] All RPM Calculations are multiplied by 10 to have a resolution of 0.1
[7] See the Load Command subsection on the next page for a description of how this command works
[8] See the Final Drive Ratio Wizard Command subsection on the next page for a description of the command's payload.

Load Command

The load command is a special command. Once received, the microcontroller will send back a command to the phone in the following format.

```
L:Units:Max_Speed:Magnets:Final_Drive_Ratio:Speedometer_Ratio:Wheel_Size_Text
```

- Ex. `L:0:120:2:1000000:18000000:P205/65R15`

Final Drive Ratio Wizard Command

This command has two different possible payloads

1. If it receives an `"S"` or an `"F"`, it will make the device enter/exit the Final Drive Ratio Calibration Mode
2. If it receives a number, it will treat the number as the average speed * 10 and use it, along with the speed readings our microcontroller obtained itself to calculate your final drive ratio. You can find the equation for this in the equations section below.

4.6.5 Debugging Commands Format Table

Here is a list of "Advanced" commands that the microcontroller will understand. These are commands that I have programmed in to use when creating and debugging the device itself. I didn't remove them as I thought they might be useful in the future for whatever reason. In order to use these commands, you must use a Bluetooth Serial Terminal app like this one: **https://play.google.com/store/apps/details?id=project.bluetoothterminal&hl=en_US**.

Command	Description	Expected Values	Example
B	Enable/Disable Debug Logs	0: Disable, 1: Enable	B:0
X[9]	Set P value in the PID Equations	0 - (2^31 - 1)	X:120
Y[10]	Set I value in the PID Equations	0 - (2^31 - 1)	Y:2
Z[11]	Set D value in the PID Equations	0 - (2^31 - 1)	Z:1

Note: PID Commands will not store the PID across reboots. If you wish to keep your PID permanently you will need to modify the source code manually. I did not implement saving PID values into permanent storage as I don't think they should be modified after production.

[9] This value is divided by 1,000 in our PID equations.
[10] This value is divided by 10,000 in our PID equations.
[11] See footnote 10

4.6.6 Programming the Values for the Microcontroller

Now all of the general commands are sent and used via the app that was created for this project; however, if there is something that the app doesn't do/implement well, you can just a Bluetooth serial terminal app to program it manually. You will then not be constrained by our app. Any Bluetooth serial terminal app will work. The one that I use was provided in the link above. I have posted the link here again for your convenience:

https://play.google.com/store/apps/details?id=project.bluetoothterminal&hl=en_US)

4.7 Equations

4.7.1 Overview

Here I will derive out all the equations that I used in the microcontrollers.

Some of the equations look weird as we are trying to avoid floating point calculations (calculations with decimal points). This is because floating points calculations take a long time. To circumvent the need for floating point calculations, I will multiply an adjustment value to the calculated values. This value will be divided out near the end to preserve as many significant figures as possible. We probably should've used a fixed-point library to keep track of this stuff instead, but the equations were simple enough and the library adds time and complexity to the runtime.

4.7.2 On the Phone

Wheel Size:

We will use the generic equation to convert metric wheel sizes to circumference in inches.

$$Circ\ (in) = \left(\frac{x \cdot y \cdot 2}{2540} + z\right) \cdot \pi$$

This equation assumes that the first set of numbers is x, second set is y, and third set is z
`(Ex. In P205/55R15 -> x = 205, y = 55, z= 15)`

From there we will convert the circumference to the value we send. Currently, we send the circumference in miles or kilometers (depending on the units selected), and then multiplied by a billion. We have the values differ here so the microcontroller does not have to deal with units at all. On the microcontroller, all calculations are unit agnostic and our variables are Speed Per Hour (SPH) instead of MPH or KPH.

The value sent for kilometers is calculated as:

$$\begin{aligned} Sent\ Value\ (km) &= Circ\ (in) \cdot \tfrac{2.54\ cm}{1\ in} \cdot \tfrac{1\ km}{100000\ cm} * 1000000000 \\ &= \tfrac{Circ\ (in)\ \cdot 2.54}{100000} * 1000000000 \end{aligned}$$

The value sent for miles is calculated as:

$$Sent\ Value\ (mi) = Circ\ (in) \cdot \frac{1\ mi}{63360\ in} * 1000000000$$

$$= \frac{Circ\ (in)}{63360} * 1000000000$$

4.7.3 On the Microcontroller

Time For 1 Sample

So, I've talked a bit about the magic numbers 65.536ms and 15.26 samples a second. These numbers are used to record the speed data of both the vehicles current speed as well as the motors current RPM. To get these values we use the system clock speed of `16,000,000 MHz`, the hardware prescaler value for Timer2 `64 Cycles/Increment`, the max value for Timer2 `256 Increments/OVF`, and the software prescaler we set of `64 OVF / Sample`

$$Sample\ Rate = \frac{16000000\ cycles}{1\ s} \cdot \frac{1\ inc.}{64\ cycles} \cdot \frac{1\ ovf}{256\ inc} \cdot \frac{1\ sample}{64\ ovf}$$

$$= \frac{15625}{1024} \approx 15.26\ (Samples/Second)$$

To get the time per sample just take the inverse of the value above. On the microcontroller we will stick with the fractional values for maximum accuracy.

The "InRatio"

Every time we calculate the speed of the vehicle, we need to use the # of Magnets, Final Drive, and Wheel Size. Since these are constants, we don't need to calculate it every time; therefore, we have a variable `InRatio` to hold the value of these 3 constants to save time during calculations.

Since our generic formula for calculating vehicle speed is:

$$Speed = \frac{\#\ of\ Magnets\ Passed}{Unit\ of\ time} \cdot \frac{1\ Rotation\ (S.S.)}{\#\ of\ Magnets} \cdot \frac{Rotations\ (W)}{Rotations\ (S.S.)} \cdot \frac{Wheel\ Circ.\ (Dist.)}{1\ Rotations\ (W)}$$

$$= \frac{Distance}{Unit\ of\ Time}$$

We can see that the last three terms in our dimensional analysis are basically our # of Magnets, Final Drive, and Wheel Size respectively. We will pull those three terms our and call it our `InRatio`[12].

$$InRatio = \frac{1\ Rotation\ (S.S.)}{\#\ of\ Magnets} \cdot \frac{Rotations\ (W)}{Rotations\ (S.S.)} \cdot \frac{Wheel\ Circ.\ (Dist.)}{1\ Rotations\ (W)}\ /\ 4$$

$$= \frac{Wheel\ Circ.}{\#\ of\ Magnets \cdot Final\ Drive}\ /\ 4$$

[12] The InRatio is divided by 4 to make sure all numbers don't get too big later on.

With the InRatio, our new equation for calculating speed becomes

$$Speed = \frac{\#\ of\ Magnets\ Passed}{Unit\ of\ time} \cdot (InRatio \cdot 4) = \frac{Distance}{Unit\ of\ Time}$$

Current Speed

The way we calculate the current speed is by taking 16 samples of data (~1 s) and using it to find the angular velocity. Once we have that all we need to do is multiply the wheel size and divide out the adjustments we made before. The samples of data tell us how many magnets have passed in each time period, and are stored in a circular array of size 16 called speedCtr.

Since this is where we calculate the actual speed of our vehicle, we will need to divide by a billion to compensate for the adjustments on our wheel size (see wheel size equation). We will also multiply the result by 10 so we have a resolution of 0.1 SPH without worrying about the decimal places.

Starting from the equation above, our Current Speed Equation (SPH x 10) becomes:

$$SPH \cdot 10 = \frac{\#\ of\ Magnets\ Passed}{Unit\ of\ time} \cdot (InRatio \cdot 4) \cdot \frac{1}{1000000000} \cdot 10$$

$$= \frac{\sum speedCtr[i]\ (magnets)}{Number\ of\ samples} \cdot \frac{15625\ samples}{1024\ s} \cdot (InRatio \cdot 4) \cdot \frac{3600\ s}{1\ hr} \cdot \frac{1}{1000000000} \cdot 10$$

$$= \frac{\sum speedCtr[i]\ (magnets) \cdot InRatio \cdot 9}{Number\ of\ samples \cdot 1024}$$

Note: The value `i` in the equation above should go from `0` to `Number of Samples - 1`.

We have left the Number of Samples as a variable instead of simplifying the number into the equation so that if you ever want to change the number of samples recorded all you have to do is change the variable `MAX_RECORD` at the top of `controller.ino` instead of having to recalculate the equation once more.

We then do some exponential smoothing to compensate for the relatively low resolution of having so few magnets for our speed sensor.

Note: Decreasing the number of samples yields less resolution to the speed of the vehicle, but will improve response times. Increasing does the opposite. You can increase the number of magnets to compensate for the loss in resolution.

Target RPM (DC Motor)

This one is fairly straightforward. When we get our speed per hour all we have to do is multiply it by the speedometer ratio to get the required motor RPM.

$$targetRPM \cdot 10 = Current\ Speed\ (SPH) \cdot \frac{Number\ of\ RPMs}{1\ SPH} \cdot \frac{1}{1000000}$$

We split the divide by a million into two divides by 1000 to ensure that the multiplication won't go out of bounds while retaining as much resolution as possible.

Current RPM (DC Motor)

The equation for this is fairly similar to the current speed equation, except we don't need to convert it to linear velocity. RPM stays at angular velocity. We will also multiply the value by 10 to get a resolution of 0.1 RPM's

$$\begin{aligned} Motor\ RPM \cdot 10 &= \frac{Number\ of\ Holes\ Passed}{Unit\ of\ time} \cdot \frac{1\ Rotation}{Number\ of\ Holes} \cdot 10 \\ &= \frac{\sum encoderCtr[i]\ (Holes)}{Number\ of\ samples} \cdot \frac{15625\ samples}{1024\ s} \cdot \frac{1\ Rotation}{20\ Holes} \cdot \frac{60\ s}{1\ min} \cdot 10 \\ &= \frac{\sum speedCtr[i]\ (Holes) \cdot 46875}{Number\ of\ samples \cdot 1024} \cdot 10 \end{aligned}$$

The array size, and subsequently, number of samples for encoderCtr is 4.

PID Equations

Our PID equations are pretty standard. Only difference is that, once again, to avoid decimals we multiply values by an adjustment. At the top of `controller.ino` you will find the pid values in whole numbers. If you scroll down to line 141-143, you can find these equations:

```
1    pid_p = KP * error / 1000;
2    pid_i += KI * error / 10000;
3    pid_d = KD * (error - oldErr) / 10000;
```

PID values are typically less than 1 so to compensate for it, we multiply and then divide the division in these lines of code will tell you the coefficients. Please note that error has an adjustment of `x10` (ex. KP is 11 so the P coefficient is actually `11 x 10 / 1000 = 0.11`)

Final Drive Ratio Equations

To calculate our Final Drive Ratios from the wizard, we need 2 pieces of information. We need the `expected distance traveled` when we dont take into account final drive, as well as the `actual distance traveled`. The reasoning behind this is if the actual distance traveled is shorter than the expected distance traveled with no final drive calculated, then we know the final drive is a ratio of the two.

$$FinalDriveRatio = \frac{Expected\ Distance\ Travled\ (w/o\ Final\ Drive)}{Actual\ Distance\ Travled}$$

Now to calculate the expected distance, we need to know how many magnets have passed over the speed sensor in a given amount of time. For our microcontroller this set amount of time is 10 seconds. When the user drives for 10 seconds, our microcontroller is recording the number of magnets that have passed into a variable named `calTicks`.

$$Exp.\ Distance = calTicks\ (Mags\ Passed) \cdot \frac{1\ Rotation\ (W)}{\#\ of\ Mags} \cdot \frac{Wheel\ Size}{1\ Rotation\ (W)} \cdot \frac{1}{1000000000}$$

$$= \frac{calTicks \cdot Wheel\ Size}{\#\ of\ Mags \cdot 1000000000}$$

To get the actual distance, we will take the value that we received from the Final Drive Ratio Wizard Command (See the Bluetooth's General Commands Format Table) from the phone. This will tell us the average speed per hour we were traveling at according to GPS. To get the total distance traveled during the 10 seconds, we just do:

$$Actual\ Distance\ Travled = \frac{Avg.\ Speed\ (Unit)}{hr} \cdot \frac{1\ hr}{3600\ s} \cdot 10\ s \cdot \frac{1}{10}$$

$$= \frac{Avg.\ Speed\ (Unit)}{3600}$$

Note: we divide by 10 at the end because the speed that we received will be multiplied by 10 to remove the decimal while maintaining a resolution of 0.1.

If we put these all together and simplify, we get the equations on the next page.

$$FinalDriveRatio = \frac{\frac{calTicks \cdot Wheel\ Size}{\#\ of\ Mags \cdot 1000000000}}{\frac{Avg.\ Speed\ (Unit)}{3600}}$$

$$= \frac{calTicks \cdot Wheel\ Size \cdot 3600}{\#\ of\ Mags \cdot 1000000000 \cdot Avg.\ Speed\ (Unit)}$$

$$= \frac{calTicks \cdot Wheel\ Size \cdot 9}{\#\ of\ Mags \cdot 2500000 \cdot Avg.\ Speed\ (Unit)}$$

We split the Wheel Size multiplication out in the code since we reuse some variables to retrieve the wheel size from permanent memory.

4.8 Issues/Things to Note

While we try our best to make it perfect, there are some issues and intricacies that we found last minute. We will try to get the issues solved, but I cannot promise as I have no control over my teammates. Not all the things in this list are necessarily issues. Some of them are things to keep in mind as they don't really matter, but in a perfect world none of these things would exist. ~ Samuel

Notes:

1. You should set your units before setting your wheel size since the calculations are done via the phone based on the units. If you set your units after then the wheel size might be in the wrong units. I have asked the app team to recalculate the wheel size on unit change, but I don't know if it has been done.
2. This might be due to my lack of experience with PID, but the motor for the speedometer seems to run slow when accelerating and run fast while decelerating. This means that if you slow down to a particular speed it will show a speed higher than what you are currently at, when you speed up the speed shown will be slower then you are actually going. This results in about a +/- 5 MPH discrepancy to the displayed speed. No matter how much I tried to tune PID I could not figure it out.
3. Since PID is dependent on timing, you cannot tune the PID while printing the debug messages (this was the issue we had when we canceled the meeting the day our beta was due). This is because when you print the debug messages, the loop slows down a lot (10's - 100's of times slower), and results in the tuned PID being way too aggressive after disabling the debug logs.
4. The delay that I added at the end of the main loop may not be necessary as the main loop runs at a fairly consistent speed. It is there because PID likes the loop to be at a constant interval in order to be stable. Removing it will sacrifice PID stability (may over or underestimate) but will improve overall responsiveness. From my experience, the impact from removing it is negligible. I kept it in there as it seemed to be the correct way of doing things.
5. During the showcase we found that if you cancel out from the wizard instead of finishing it, the microcontroller gets stuck in that mode. I have asked the app people to send D:F and P:0 when they cancel out of the wizard, but I don't know if they will get to it. For now, if it gets stuck in that mode just press the reset button on the microcontroller to have it restart and everything should work fine.
6. Right before the showcase we found out that the app will crash if the data sent from the microcontroller to the phone upon connecting is invalid/corrupted. This typically happens if the app crashes while sending the data to the microcontroller but there are other reasons why it could happen. To fix this you will either need to program it manually using a Bluetooth serial terminal app, or just run the reset script that I will include in the repository. I have asked the app people to simply ignore corrupted data so this won't be an issue, but again I don't know if they will get to it.
7. Since we only have a maximum of 4 magnets for each speed sensor the resolution of the speed is

somewhat estimated. We recommend for best performance to put 4 magnets if you are reading the speed on the axle or after the final drive gearing. If it is before the final drive gearing then having less magnets are fine.

8. If you decide to source your own magnets please do not put more than 20 magnets if reading from the axle, and no more than 4 if reading before the final drive gear change. This is due to the fact that the microcontroller can only handle values up to a certain size. If the numbers get too big weird things happen.