

- change [otis_arduino](#) so it works with Arduino MKR WiFi 1010 (boards are provided)
 - BluetoothSerial.h library does not work with MKR. MKR can only use ArduinoBLE library that does not provide serial data transfer from what I can tell. Commented out the bluetooth component the otis_arduino software for now, until we can find a better solution for wireless data transfer.
 - Considering a webserver to collect data transferred from the MKR through wifi.
 - ledcwrite() is an ESP32 specific PWM function, and does not work with the MKR. The only alternatives are manually modifying register values to allow for the PWM you want or SAMD21 turbo PWM library. According to their Git, https://github.com/ocrdu/Arduino_SAMD21_turbo_PWM. The library is “untested” with the MKR series that we are using, and I had to modify their header file to add “ARDUINO_SAMD_MKRWIFI1010” so that the Arduino IDE would allow this library to be used with our brand of Arduino. I was able to use the library to output a 30KHz PWM, and tested the PWM with my scope to make sure it output what I needed.
- test it on the robot and see if it is stable and can be controller

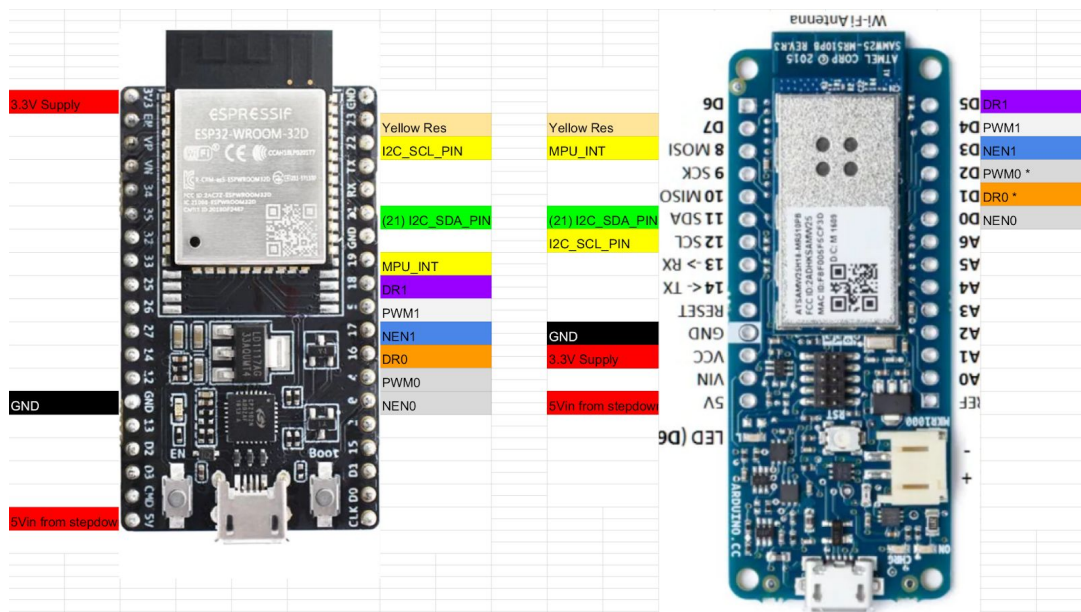


Figure 1: Wiring Diagram for board modification.

- convert the PID control defined in [otis_arduino](#) to a Rust library
 - First I attempted to use the pid_control crate for this feature, but quickly started running into issues getting the allocator to work with the structs. Since this library has to be FFI, and has to be embedded I was significantly limited by the libraries I could use. I finally got to a point where I could compile the Arduino code using the Rust library with a Rust allocator. This doesn't work, but it's worth documenting:
 - I used the cortex-m quickstart library to generate a generic starting place for my PID library. This adds features to the build function to specific the build target, and the locations and sizes of memory on the board. The build target for the MKR1010 which has a cortex-m0plus is **thumbv6m-none-eabi**
 - I had to use the memory allocator **alloc-cortex-m** to allocate memory for the structs I was trying to create based on the pid_control Rust library.
 - `#![feature(alloc_error_handler)]` because the allocator needs an error handler that needs to define what happens in an Out of Memory condition. The error handler function can be found in the allocator.rs example of the cortex-m0+ quickstart guide. There are several other requirements necessary to have the allocator properly work that are described there.

- Making the library compatibility with C was a whole challenge on it's own. `#![no_std]`, and `#![no_main]` had to be used. I also utilized **cbindgen** to create the necessary C header files. I also had to create structs and pass them through as raw pointers from the rust end, then those structs had to be sent back to rust to be cleared out.
- At this point the program compiled using the rust library and alloc, and I was able to upload it with the arduino C++ code, but it would still not run after being uploaded.
- I modified the arduino rust library to not use any memory allocs, and that compiled and worked.
- I had to modify the rust library several times until I could get the Rust library output the same values as the old C++ library with the same inputs. I'm also using Arduino to provide the `millis()` time values, I'm not entirely sure if this is the best approach and if there is a way to grab the time from inside rust.
- 4. recompile project with the Rust library instead of C/C++ library and test the robot's behavior'
- Testing the robot I ran into multiple issues, I had a constant wobble while the robot was trying to correct itself. And often the robot would overshoot, and hit the other side. I had to disable a feature with the library that tracked the integer value to correct the overshoot. And I started to adjust the PID values to recalibrate it, but ran into problems with the center of mass being off. I also started running out of battery charge, and could not test it further.

One problem I've been consistently having with the robot is the IMU values hanging up, or freezing. Occasionally that will occur when the motor starts, so I added multiple decoupling capacitors at the voltage supply from the battery, and around all the components. Even without the motors I would get occasional hang up just from running the basic PID program without any communication.

I added 2.2k pull up resistors to the I2C SCL and SDA lines to improve the square wave since we're running the program at 400Khz, that didn't seem to make a massive difference, but the square wave looked a bit better.

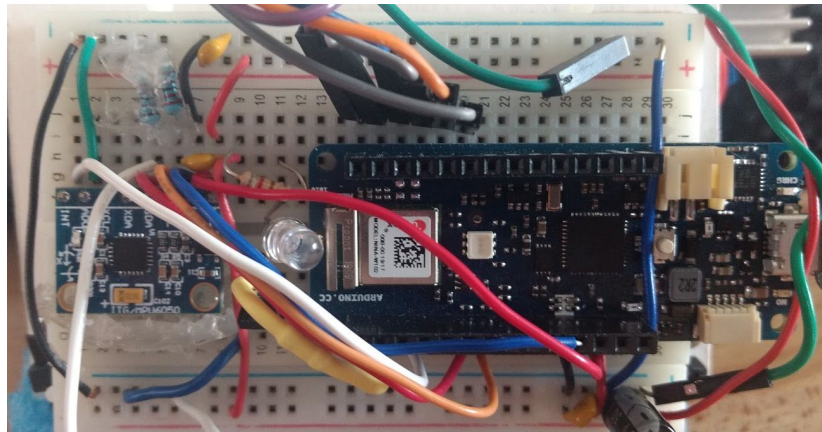


Figure 2: New board replacement.

I tried getting the new version of the MPU6050 library, and that seems to help improve the data transfer speed significantly. I had to modify the new MPU6050 library to work with our Cortex processor by adding:

```
#ifdef ARDUINO_ARCH_SAMD
```

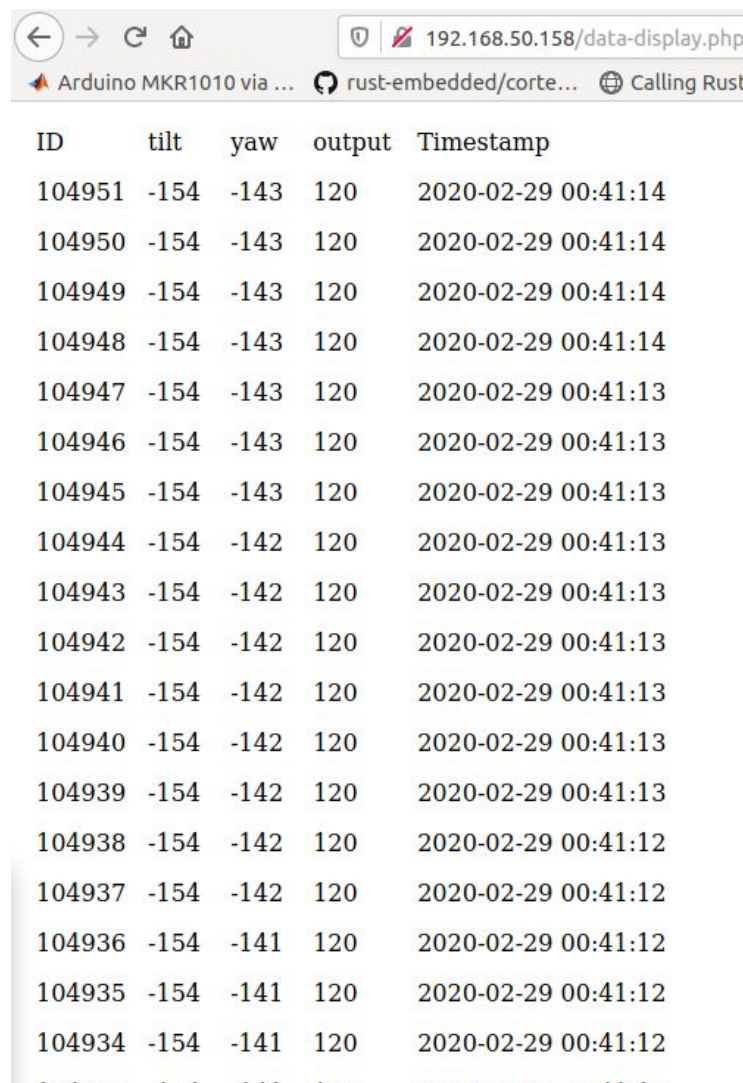
```
#define BUFFER_LENGTH SERIAL_BUFFER_SIZE
```

```
#include <avr/dtostrf.h>
```

```
#endif
```

I tried the new method of “**mpu.dmpGetCurrentFIFOPacket(fifoBuffer)**” to grab data from the buffer, but this results in a lot of issues. The MPU would consistently hang up after running for 10 seconds to 5 minutes, and I did not realize that this code change was the problem until much later on. I resorted to using the old method of grabbing data from the fifo, and that was able to run about +7 hours without hanging. So I believe the IMU hanging issues might finally be resolved on my end.

At this point I thought it would be a good idea to add some form of data communication with the PC since we will need to be able to compare our data. I thought it would be a good idea to write the data to a web server using HTTP post, before I had a good idea of what that really means. I created an Apache2 + MySQL server, and handled the data using a couple PHP scripts.



ID	tilt	yaw	output	Timestamp
104951	-154	-143	120	2020-02-29 00:41:14
104950	-154	-143	120	2020-02-29 00:41:14
104949	-154	-143	120	2020-02-29 00:41:14
104948	-154	-143	120	2020-02-29 00:41:14
104947	-154	-143	120	2020-02-29 00:41:13
104946	-154	-143	120	2020-02-29 00:41:13
104945	-154	-143	120	2020-02-29 00:41:13
104944	-154	-142	120	2020-02-29 00:41:13
104943	-154	-142	120	2020-02-29 00:41:13
104942	-154	-142	120	2020-02-29 00:41:13
104941	-154	-142	120	2020-02-29 00:41:13
104940	-154	-142	120	2020-02-29 00:41:13
104939	-154	-142	120	2020-02-29 00:41:13
104938	-154	-142	120	2020-02-29 00:41:12
104937	-154	-142	120	2020-02-29 00:41:12
104936	-154	-141	120	2020-02-29 00:41:12
104935	-154	-141	120	2020-02-29 00:41:12
104934	-154	-141	120	2020-02-29 00:41:12

Figure 3: Data write to local network

This worked well as a demo, and the processor was able to write data, but I was running into multiple hang up issues on both the IMU end, and the server end. So it was essentially impossible to run the robot without some form of hang up, which is not acceptable. I suspect the issue with this method is that the HTTP header is too long, and it's too time consuming for the processor to read and write to the client while trying to

handle the IMU interrupts. I attempted to use just a simple TCP socket to read data directly to linux using 'nc', and that doesn't slow down the arduino processor much at all, but I'm currently having issues with the program disconnecting. I was hoping to create a program in Rust that reads the data from TCP, and writes it to the SQL server. I was able to create a preliminary program that communicates and reads data, but I have not fleshed it out yet.

Summary:

Rust PID library is working, IMU hand up issues seems to be resolved, and I really want to try to write the data to a web server for comparison because it would be a very convenient way to access the data, but it might be too time consuming at this point.