Project Report

Cube Visualizer (Orientation Visualizer)

Cameron Oehler

200384643

ENSE 481

Karim Naqvi

April 11, 2025

Design

For this project, I have designed a system to visualize the orientation of a hardware circuit. The system consists of three main components. The first component is a hardware circuit consisting of a STM32F103 board and an Adafruit 9DOF board (previously also included a Adafruit Bluefruit SPI board). The second component is a Python webserver for interfacing communication between the hardware and the visualizer. And the third component is a Godot project which polls the Python webserver and uses the returned information to manipulate the orientation of a 3D rendered model of the hardware circuit.

Hardware Component

At its core, the system is a RTOS (running FreeRTOS) with two tasks (originally there were going to be three tasks).

The first task is for setting up and polling the IMU. At a high level, this task first sends some commands to the IMU to enable and configure it. It then continuously sends commands to the IMU requesting data, performs calculations on that data to get roll, pitch, and heading in degrees, and then adds that data into a queue. This happens every 50 ticks. If the queue fills up, the task will block and wait for space to become available. Communication with the IMU is done over I2C. To send data to the IMU, the HAL I2C commands were used. I created several wrapper functions around these commands in order to perform error checking, send multiple bytes (register & value), and easily receive single bytes. The IMU itself consists of two separate chips. The L3GD20H contains a gyroscope and while methods to configure and read this chip were implemented, they were not utilized in the final version of the hardware component. The LSM303DLHC contains an accelerometer and a magnetometer. Both of these are used to determine the orientation of the IMU. The accelerometer can be used to calculate the roll and pitch of the device as the acceleration of the device changes in the x, y, and z axis when the device is rotated in the roll and pitch axis (due to acceleration from gravity). The accelerometer on its own cannot be used to determine heading because a rotation in the heading axis does not change the acceleration felt by the x, y, or z axis. For this reason, we also use the magnetometer along with the accelerometer data to determine the heading of the device. The formulas I used to calculate the final roll, pitch, and heading were adapted from the Adafruit 9DOF library.

The second task is for controlling the CLI (command line interface). This task first sends data to the USART clearing the screen and printing the prompt. It then continuously polls the USART for a single character. It prints that character back out to USART and stores it in a buffer. When the character received is a ‘\n’ or ‘\r’ (newline) the data buffer is tokenized, processed, and ultimately runs the command specified before clearing the

buffer and starting the whole process again. The number of characters stored in the buffer is also saved and is used to enable backspace capabilities by decrementing the count. The cli interaction with the USART is done in a blocking mode, simply because it is running on an RTOS and there would be no reason to use interrupts (although functions for this were written and reside in the codebase). The following commands were implemented in the cli:

- help – prints a list of available commands
- printBinInfo – prints the current version number (containing date and git hash) and build date of the binary
- clear – clears the screen
- orientation – reads 3 values from the IMU queue (roll, pitch, and heading) and prints them out
- stream – continuously reads 3 values from the IMU queue and prints them out in a JSON format. This runs forever and prevents any further cli activity (this method is intended to be used with the Python webserver to get IMU data). This happens every 10 ticks (or when data is available)

Originally the plan was to transmit the IMU data over a BLE (Bluetooth low energy) connection to the visualizer. Unfortunately, I was unable to get communications to work properly with the Adafruit Bluefruit SPI board. However, the code for this was written and does reside in the code base. The code written was based on a few sources including the Adafruit Bluefruit library and the Adafruit SDEP (Simple Data Exchange Protocol) reference. The Bluefruit SPI board uses the SPI protocol for communication, but uses another layer on top of that called SDEP. This protocol consists of sending packets of data no more than 20 bytes in length. The first byte of every packet is a message type (command, response, alert, error, not ready, or data read exceeded). In a command or response message the next byte is a command ID. The receiving device (when sending a command message) uses this ID to determine what the sender is trying to do (ex. Send a command, send text, etc.). In a response message this ID will be the same as the ID in the command that generated the response (this allows a sender of a command to understand what response matches what command). The message ID always consists of 2 bytes (these are sent in little endian format). The next byte of data is the payload length. A payload can be up to 16 bytes long (20 bytes total – 4 bytes header = 16 bytes payload). Bits 0-3 of the payload length are used to specify the length of the payload. Bits 4-6 are unused. And bit 7 (more data bit) specifies if the message is made of multiple packets. The SDEP specifies that when the more data bit is set, the receiver must continue to read packets until the bit is not set and combine the payload of all received packets. While implementing this protocol I discovered that the IMU would begin responding to my packets while I was still sending them. This led to implementing a ring buffer system which would store received bytes while sending bytes.

This would then be pulled from first before more data was pulled from the BLE via SPI. Unfortunately, I was unable to establish communication with the Adafruit Bluefruit SPI and had to pivot the communication for this project to USART.

Python Webserver

The Python webserver was implemented in order to proxy communication between the serial interface of my computer (which was connected to the USART of the STM32F103) and the Godot game engine. The Python program uses the packages flask (for the webserver), pyserial (for the serial communications), and threading (to run the serial communications concurrently with the webserver). Upon running the program the first thing it does is start a thread for the serial communication. This then sends the command 'stream' to the STM32 which causes it to begin responding with the IMU data. The Python server then continues to read this data and update a global 'data' variable. After beginning the serial communication thread, the flask webserver is started. When a http GET request is sent to '/' on localhost port 8080, the webserver responds with the IMU data.

Godot Visualizer

Godot is a program primarily used for developing games. However, its 3D rendering engine and ability to make HTTP requests lends itself well to this project. Using a few basic shapes, I modeled the hardware component of this project. A script then runs over and over which makes a GET request to the Python webserver. The response from the webserver is then decoded and used to set the x, y, and z rotation values of the model. This produces a system which matches the rotation of the hardware to the visualization on the screen.

Testing

Before having all the components together, each component was tested independently. This began with testing the CLI and USART communications such that known data being sent in was received correctly, parsed correctly, and the expected data was returned correctly. This included testing that the data being sent to the Python server was received error free. This was tested by sending a known string and making sure it matched in Python. This was also tested by watching the Python server for data parsing errors (if there were none, then the received data was valid JSON which meant it was unlikely to contain any other errors).

To test the IMU I tested orienting the hardware in various orientations. I then checked the computed data (roll, pitch, heading in degrees) from the IMU and made sure the measurements were consistent with reality. I discovered that in general the IMU was

fairly accurate but did sometimes jump to a random value that was inconsistent with reality. This could be due to any number of things. This could be due to various inaccuracies in the accelerometer and magnetometer. It could also be due to magnetic interference that is messing with the magnetometer reading. Either way, the reading is consistent enough to use for this project.

When all the components of the project were put together, everything was tested together to make sure that the orientation of the visualization matched the orientation of the hardware. This was done the same way as when I was testing the IMU on its own except now I was checking if the visualization orientation was consistent with reality,

Usage

To use the project, build the program in the 'cube' directory using the STMCubeIDE and flash the STM32F103 board. Attach the board to the computer via USB and find the communication port assigned to the device. At this point, cli communications can be established. If you want to setup the visualizer then you will need to install the required dependencies for the Python server. This can be done using the requirements.txt file located in the 'visualizer_interface' directory. More information is available in the readme file in the same directory. Then in the main.py file change the variable 'COM_PORT' to the communications port of the STM32 board. The python server can then be run by running main.py. If any data has previously been sent to the STM32 before starting the server, it may need to be reset so the stream command is recognized. This should start a webserver on the port 8080. Finally with the Godot engine installed, the project in the 'visualizer' directory can be opened and run. If everything worked correctly, you should see a rendered version of the board and it should match the orientation of the hardware. If it doesn't match the heading of the hardware, then the node RotationOffset can have its y rotation changed to compensate.

The flow of data through the system is IMU -> Queue -> CLI -> USART -> Python Webserver -> Godot Visualizer