WIRELESS GLOVE CONTROL UNIT

Pratyush Ashok Anand (pa29)
Vikram Chakravarthi (vikram5)
Ansh Tulsyan (ansht2)

May 12th, 2025

ECE 395: Advanced Digital Projects Lab
University of Illinois at Urbana-Champaign

Terms and Keywords

ADC – Analog-to-Digital Converter

BOM – Bill of Materials

BOOT0 – STM32 Boot-Mode Strap Pin

BTLE – Bluetooth Low Energy

CS – Chip-Select (active-low SPI enable)

DFU – Device Firmware Upgrade (USB bootloader mode)

DRC – Design-Rules Check (PCB CAD verification)

EN / CHIP_PU – ESP32 Enable Pin

ESP32 – Wi-Fi / BLE Microcontroller Module

FT232 – USB-to-UART Bridge IC

GPIO – General-Purpose Input / Output

IMU – Inertial Measurement Unit (ICM-20948)

I/O – Input / Output (digital or analog interface)

LDO – Low-Dropout Linear Regulator

LED – Light-Emitting Diode (power indicator)

LM324 – Quad Operational Amplifier

MCU – Microcontroller Unit (STM32L031)

MOSFET Level-Shifter – 2N7002 bidirectional translator

PCB – Printed Circuit Board

PWM – Pulse-Width Modulation (optional haptic/LED drive)

RX / TX – Receive / Transmit data lines (UART)

SPI – Serial Peripheral Interface (sensor bus)

SWD – Serial-Wire Debug (ARM programming interface)

UART – Universal Asynchronous Receiver/Transmitter (debug serial)

VR – Virtual Reality hand-tracking application

Abstract

We attempted to build a wireless gesture controlled glove unit that would enable us to interact with the VR world, effectively controlling and manipulating simple 3D models created using the Unity software. We attempted to do so using a combination of sensors (flex sensors, IMU) and wireless transmission protocols and devices (ESP32 microcontrollers, ESPNOW) and were largely successful in our attempts, as our final prototype managed to successfully maneuver a 3D modeled hand in spite of some minor hardware issues that we faced during the demo.

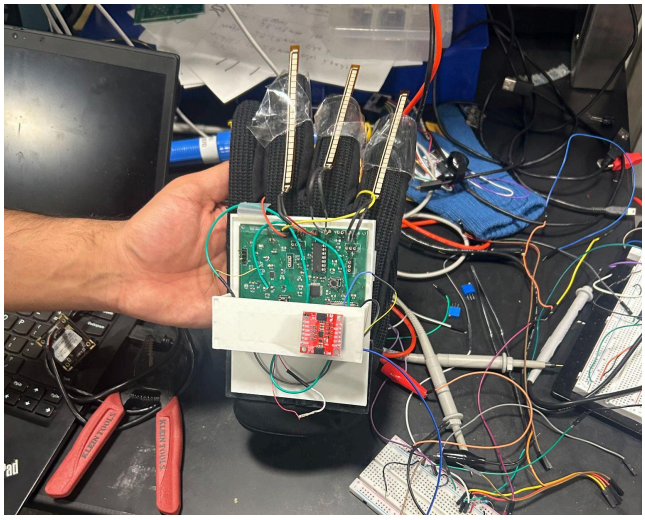


Figure 1: Image of our final wireless glove control unit prototype (any straggling wires are unrelated to the project

Overview of Project

In order to better understand and visualize the various stages involved in the creation of this project, we designed a comprehensive block diagram to explain the various stages involved in our project (**See Figure 2**).

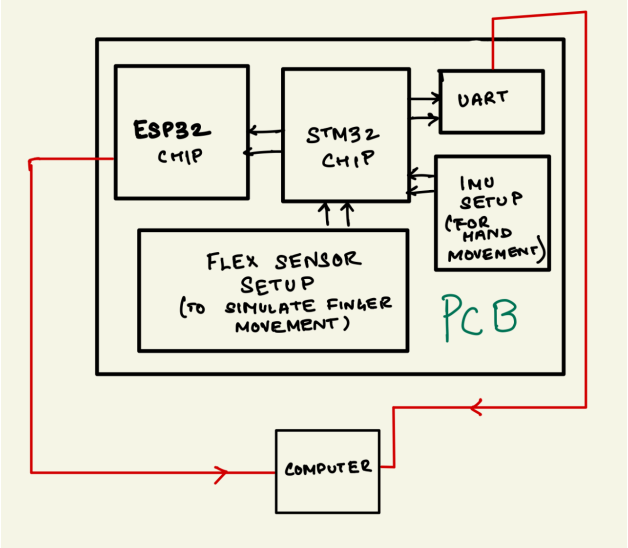


Figure 2: Simplified Block Diagram of our Project

Inside our glove PCB the flex-sensor array and the hand-mounted IMU feed raw analog data into the STM32, which digitizes and bundles everything into a single UART stream. That stream is handed off to the on-board ESP32, which relays it wirelessly to the computer running the VR demo (red path). Commands or firmware updates can travel the same route in reverse, so the ESP32 acts as the wireless bridge while the STM32 does all the sensor crunching.

The computer receives the data from the PCB using ESP-NOW wireless technology, and we use a 'receiver' program in conjunction with the series of python code in order to receive, process and subsequently transmit this code to the Unity software package that we have used to build our 3D model. We then manipulate the .cs (unity files) in order to model the hand movements as per our requirements.

Critical Components/Chips used in this project

STM32 Microcontroller- STM32L031K6T7
ESP32 Microcontroller- ESP32 WROOM E
IMU- ICM20948 by Invensense
Flex sensors-

Set up of the STM32L031K6T7 microcontroller

For the purpose of our project, we chose the STM32L032K6T7 microcontroller that we had previously used in the initial exercises we did in class. This was mainly because (a) this STM32 chip was readily available in the market and very easy to solder (b) Having worked with this STM32 MCU we felt that it was the natural choice to interact with the huge amount of code we

had written prior to working on the PCB. However, for future reference, we do recommend using a newer, and possibly more powerful STM32 board, as this heavily depends on the components that you are interfacing with the STM32. Eg: Some newer IMUs require us to use more powerful STM32 boards in order to establish effective communication/data transfer, and this could mean that the STM32L031K6T7 is not necessarily the best choice.

That being said, our STM32 is sort of the CPU of our wireless glove control unit, as it is responsible for converting all the raw data received as the input into the processed output that is fed into the ESP32 and subsequently the computer for the VR portion of the project. **Figure 3** denotes the set-up of the STM32 microcontroller in the PCB layout design.

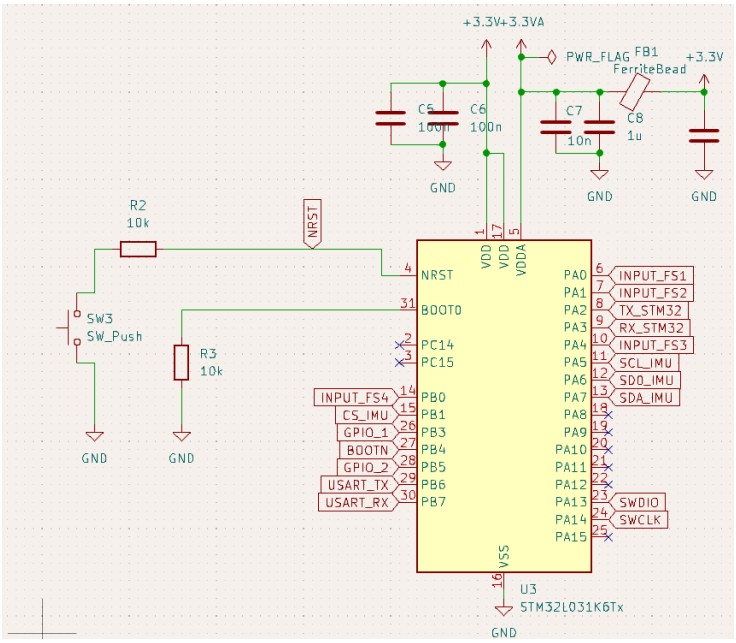


Figure 3: Diagram of the STM32L031K6T7 microcontroller as designed for our PCB

The two 10 kΩ resistors set the STM32's default states: one pulls **NRST** high so the MCU runs until you press the reset button, and the other pulls **BOOT0** low so it boots from user code. The cluster of small ceramic capacitors (100 nF, 10 nF, 1 µF) sits close to the VDD pins to smooth out supply noise across a wide frequency range, while the ferrite bead isolates the analog 3.3 V rail from digital switching spikes. Together, these passives give the chip clean power and predictable start-up behavior.

The rest of the pin assignments are as follows:
Pins (exposed as headers) used to program the STM32 microcontroller using the ST-LINK debugger:
SWDIO (PA13/23), SWCLK (PA14/24), NRST (4), 3.3V (external to STM32), GND
The following pins served as the ADC_INPUTS from our flex sensor subcircuits:
INPUT_FS1, INPUT_FS2, INPUT_FS3, INPUT_FS4
The following pins served as UART transmission pins:
RXSTM32, TXSTM32

The following pins as the IMU input pins:
SCL_IMU, SDA_IMU, SD0_IMU, CS_IMU

All the listed pin assignments will be examined in significant detail in the following sections.

UART

Although not a major requirement for the end-goal of our project, which is to able to control a 3D VR model using the wireless glove unit, the addition of a UART port, allows us to to easily the see the output of the processed data that is transmitted from the STM32 to the ESP32 through the TX pins. Likewise we may also check the RX/TX pins of the ESP32 in order to see if the data is being received properly from the STM32 output pins.

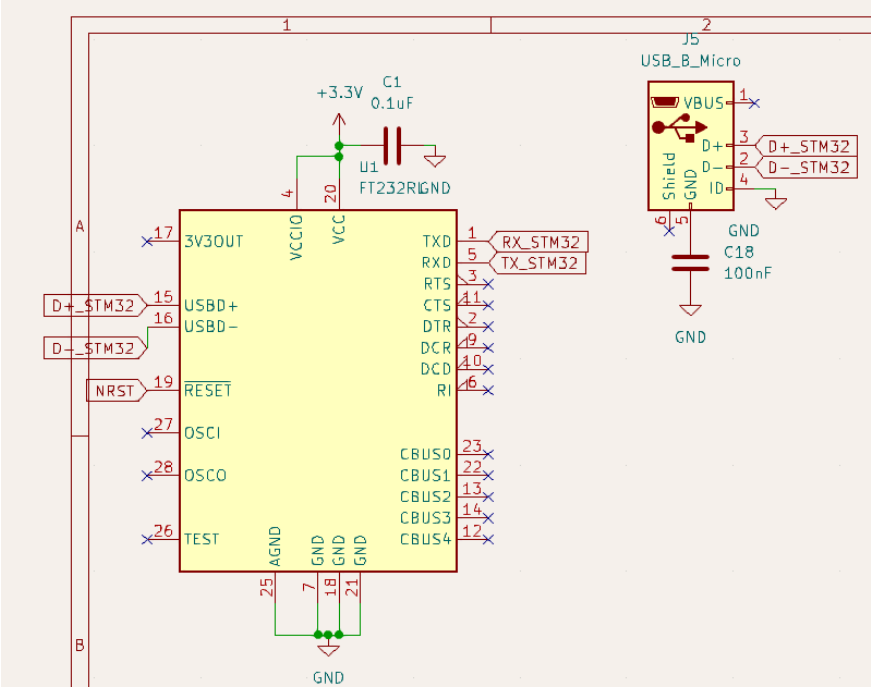


Figure 4: Diagram of the STM32 to UART IC connection, we also use a USB-micro-b connector

In Figure X, The micro-B connector (J5) brings the PC's USB signals and 5 V VBUS onto the board. The D+ and D− pairs are routed two ways: straight into the STM32 for native DFU support and into the FT232R bridge (U1), which converts them to 3.3 V CMOS-level TXD/RXD that cross over to the MCU's USART pins for printf-style debugging. U1's core and I/O rails share the board's 3.3 V supply; a 0.1 µF decoupling cap (C1) sits beside its VCC/VCCIO pins, while a 100 nF cap (C18) on VBUS tames cable-plug surges. Hand-shake lines (RTS/CTS, DTR, etc.) are left unused, though DTR could be jumper-linked to NRST if automatic bootloader resets are ever desired. In short, this sub-circuit lets any laptop recognize the glove as a standard COM port, making firmware uploads and live serial logging plug-and-play.

Finger-based gesture control

Overview

We attempted to accurately mimic finger movement in the VR world using an op-amp based flex sensor circuit, as the bending of these flex sensors would effectively manipulate the input voltage received at the ADC inputs of our STM32L031K6T7 chip. Subsequently these inputs will be converted to Digital logic through the MCUs in-built ADC converter, and wirelessly transmitted from our PCB board to computer using our ESP32 chip and ESPNOW protocol. This section will further expand upon this process in much more detail.

Hardware Design and Implementation

To verify that our flex sensor was functioning correctly—and to calibrate its output so it cleanly maps finger positions into the VR environment—we designed a dedicated signal-conditioning stage. After weighing several options, we settled on an op-amp-based adjustable buffer circuit (**Figure 5**). An adjustable buffer combines the ultra-high input impedance of a voltage follower with a tunable gain stage, so it neither loads the flex sensor nor distorts its native resistance–to–voltage relationship. This topology lets us:

- **Have no load on the sensor.** The buffer's huge input impedance means it "looks but doesn't touch," so the flex sensor's reading stays accurate.
- **Turns tiny swings into full swings.** Adjustable gain stretches the raw 0.5 V–2.5 V signal to fill the microcontroller's 0 V–3.3 V range, giving us more resolution.
- **Cuts noise and drift.** A clean, low-noise op-amp plus a small filter capacitor keeps RF chatter and slow voltage drift out of the data. This is ideal especially given that we will be implementing this on the PCB.
- **Easy hardware tweaks.** Tiny trim pots (or digital resistors) let us re-zero or re-scale any glove on the bench—no code changes required.

Hence, this simple op-amp stage gives us a clean, full-scale voltage that the VR software can use immediately for smooth finger tracking.

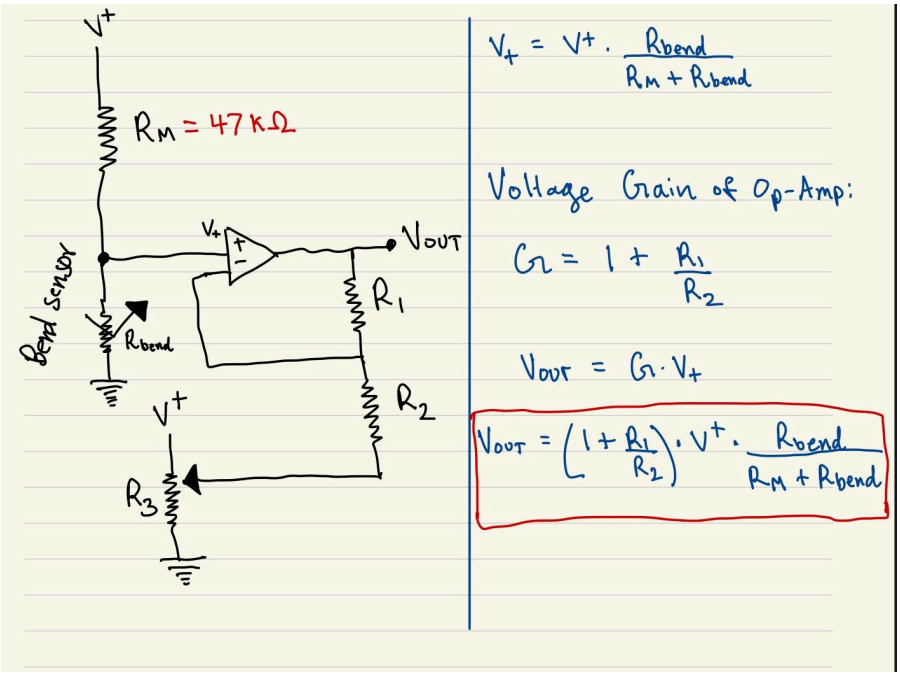
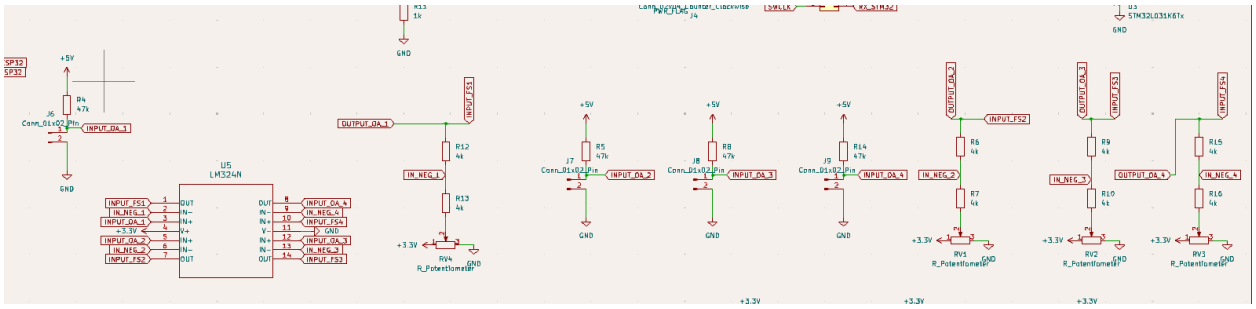


Figure 5: Schematic and relevant calculations used for our Flex circuits hardware portion

Figure 5 shows that our key variable is the buffer's output voltage, **V**, because that is exactly what the STM32 will digitize. Using the standard gain formula, we swept several resistor pairs and found that R_1 = 5 kΩ and R_2 = 1 kΩ stretch the flex-sensor's raw swing into a clean 0.8 V – 2 V window—well inside the MCU's 0–3.3 V ADC range while leaving headroom for noise or drift. To make field calibration painless, we inserted a high-value 3296-series multi turn potentiometer in the feedback path. A few screwdriver turns let us fine-tune gain so the voltage change tracks finger bend precisely, without touching firmware.The exact op-amp model isn't critical here; any rail-to-rail device with low offset and bandwidth above a few hundred kilohertz will maintain accuracy. We chose a readily available, low-noise CMOS op-amp (LM324 op-amp) to keep BOM cost down and ensure rock-solid readings, and these op-amps are readily available in most university labs as well.

PCB Implementation



Using a single LM324 quad op-amp instead of four separate single-channel devices keeps the design smaller, cheaper, and more uniform: one IC footprint frees PCB area and reduces routing clutter, one set of decoupling capacitors simplifies power integrity, and a single part number

trims BOM cost and assembly time. Because all four amplifiers share the same silicon die and temperature, their gains and offsets track closely, giving more consistent sensor readings across fingers. Fewer supply stubs and analog traces also lower the risk of crosstalk or noise pickup, so the overall glove electronics stay compact, economical, and electrically cleaner.

Software Description for Flex Sensors:

The software component of our glove interface played a crucial role in processing analog flex sensor signals, managing IMU communication, and transmitting synchronized sensor data wirelessly. This section outlines the software modules we implemented and the associated challenges we encountered with regards to the flex sensors.

STM32CubeIDE Configurations:

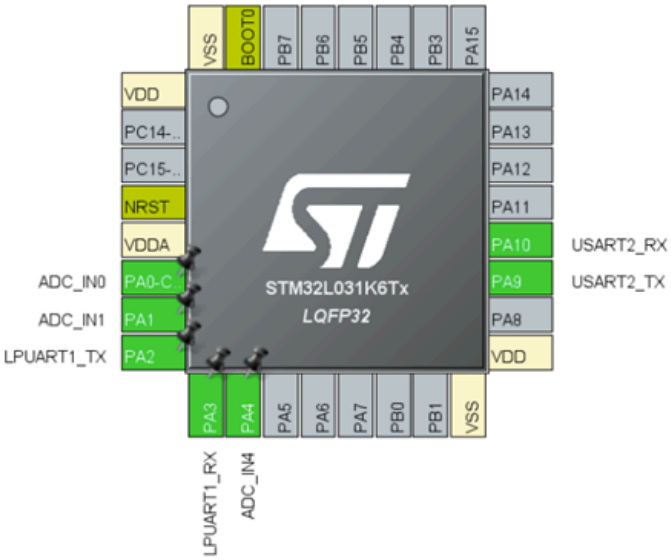


Figure 6: The pin-out Configuration of the STM32 on the CubeIDE

The above set up for STM32 refers in particular to the flex sensors.
We set up UART displaying the flex sensor values. These values ranged roughly from 1200 to 3600. They were eventually scaled according to the voltage values in the Serial Data to Python Conversion Section which will be touched upon later.

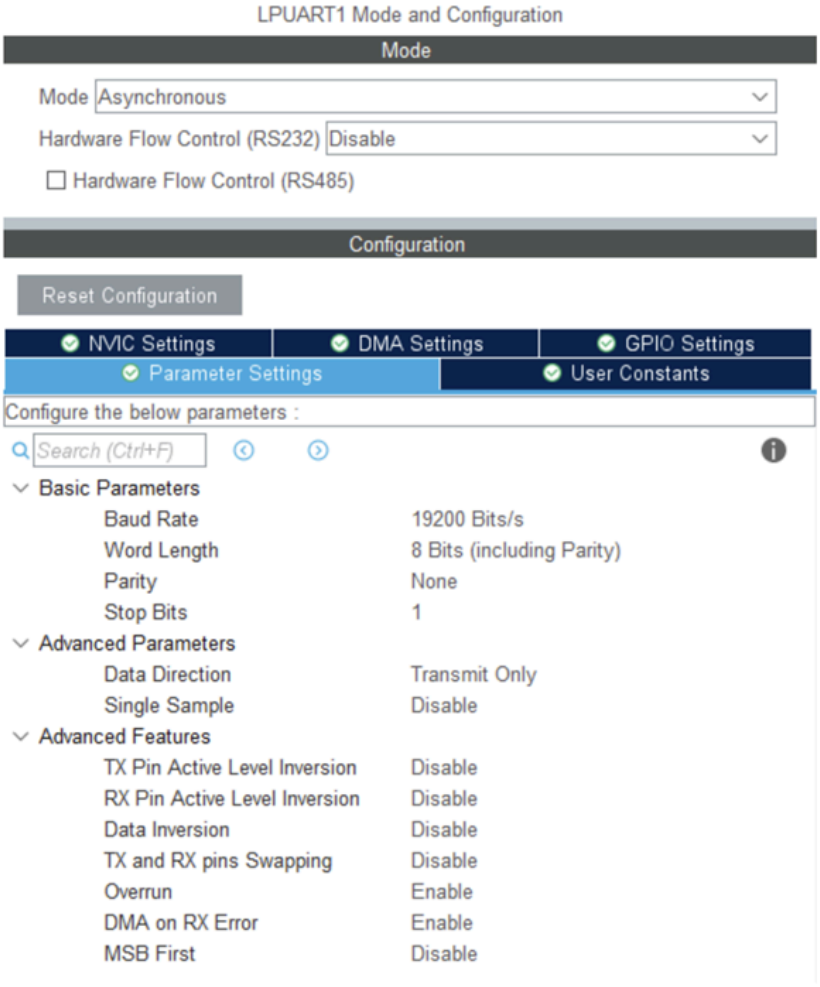


Figure 7: UART Configuration for Flex Output

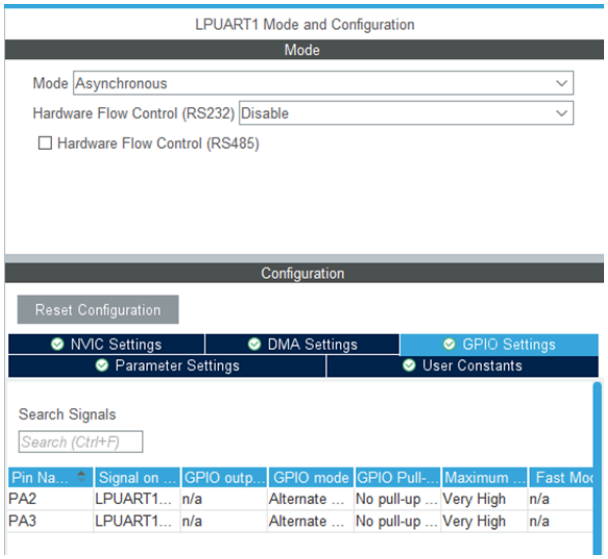


Figure 8: Mode and Configuration for UART

Finally, the ADC values were collected from three separate pins as we used 3 flex sensors in the final configuration.

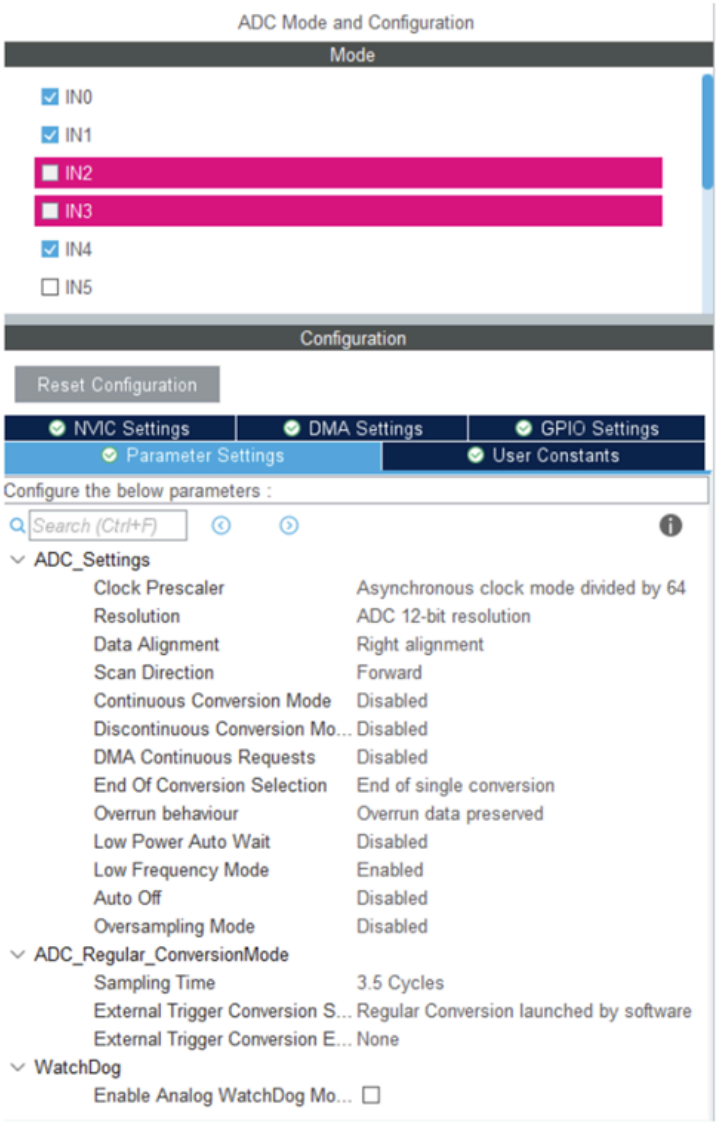


Figure 9: ADC Mode and Configuration for the Flex Sensors

Serial Data to Python Conversion for Calibration:

To calibrate the flex sensors, we transmitted raw ADC values over UART from the STM32 to a PC running a Python script. This script parsed the incoming ASCII-encoded values, visualized the flex data in real time, and helped us establish individual threshold ranges for each finger. These thresholds were later mapped to finger joint angles in Unity to animate the virtual hand.

Challenges:

- Ensuring consistent termination characters (\n\r\0) was essential to parse values correctly.
- Occasionally, overflowed serial buffers caused partial or corrupted data, which we mitigated by increasing delays between transmissions and using a custom protocol with header bytes.

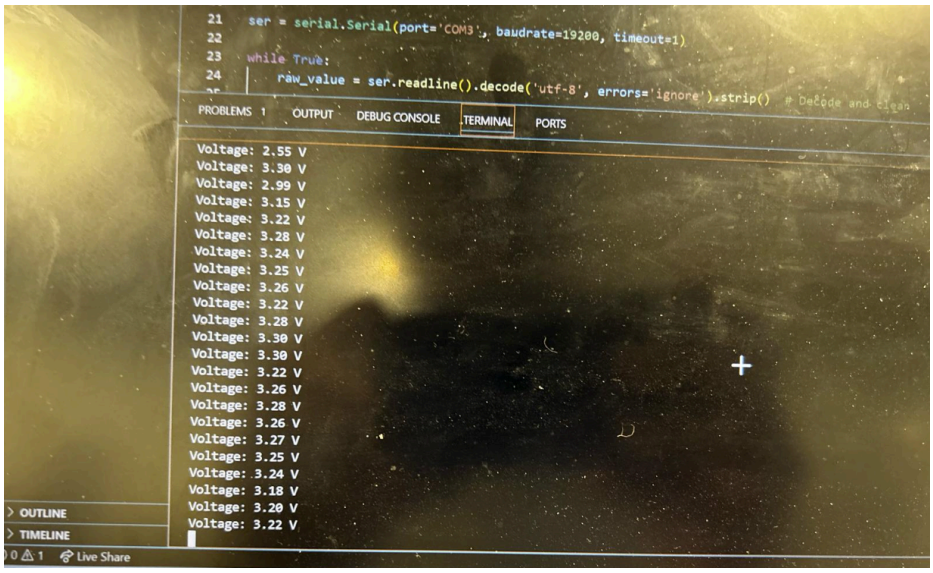


Figure 10: Output of the Python Script Displaying Flex Output in Voltages

Adapting Code to Handle Multiple Flex Sensors:

Initially, the system was designed for a single sensor. We extended it to support three flex sensors by using a looped HAL_ADC_PollForConversion() structure that iteratively triggered conversions on each ADC channel. Data from all three channels was sequentially transmitted over UART with minimal delay between them.

Modifications:

- Adjusted DMA and ADC configuration to scan multiple channels.
- Introduced batching and consistent delimiters in UART transmission for reliable parsing

ADC Issues (formatting, delays and buffers):
The onboard ADC initially returned unstable values due to noise and inadequate sampling time. We addressed this by:

- Switching to a sample time of 3.5 cycles, which balanced speed and stability.
- Using oversampling and averaging techniques in software to smooth input.
- Adding delays (HAL_Delay(10)) between ADC reads to reduce buffer overflows.
- Avoiding DMA due to complexity and opted for a simpler polling-based approach, which gave us more control during debugging.

Overview of the Flex sensor set-up (post implementation)

Overall, we believe that the implementation of the flex sensor was rather successful, and the PCB design worked as intended almost immediately (following our first attempt soldering the components onto the board). The only improvement that we could possibly make is to the software component, where we could've possibly used an interrupt based approach in order to save CPU cycles. While this is useful in numerous memory-constrained applications, we feel that it is not really a necessary addition in the context of this lab.

Overview of the IMU subcircuit

To capture real-time palm motion, we added an IMU to the glove. The sensor streams orientation and acceleration data to the STM32 microcontroller over SPI, a well-documented bus that provides the high speed and continuous throughput our application needs. By leveraging SPI we can reliably sample, process, and fuse the IMU readings with the flex-sensor data, giving the VR software an accurate picture of hand position at all times.

Hardware implementation of the IMU Subcircuit

For the purpose of our PCB design, we solely implemented the SPI-specific hardware in order to better fit space constraints and better mitigate possible noise issues from using too many components (**see Figure 11**). These design choices were made with respect to the following document provided by Sparkfun (the breakout board we used for testing purposes):

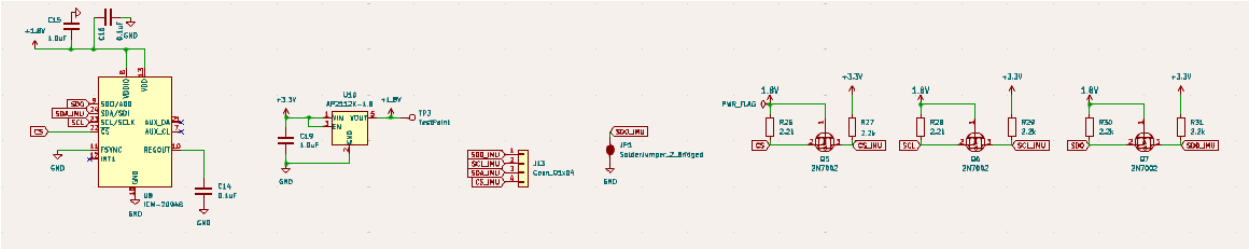


Figure 11: Image of the SPI-based ICM20948 IMU subcircuit

The ICM-20948 (U8) is powered from its own 1.8 V rail generated by an AP2112K-1.8 LDO (U9). Two small ceramic capacitors (C19 1 µF on the regulator, C15/C16 0.1 µF on the sensor)

sit within a few millimetres of the VDD / VDDIO pins, following the “typical operating circuit” in the data-sheet, to soak up fast current spikes and keep the MEMS cores quiet.

Because the STM32 runs at 3.3 V logic while the IMU speaks 1.8 V, each SPI signal (CS, SCLK, SDI, SDO) passes through a discrete 2 N7002 MOSFET level-shifter. A 220 Ω series resistor on the 1.8 V side tames edge ringing, while a 2.2 kΩ pull-up on the 3.3 V side lets the MOSFET’s body diode and channel translate edges in both directions. The result is a fast, bidirectional interface that meets the IMU’s 7 MHz SPI timing spec without stressing either device.

SPI communication

SPI (Serial Peripheral Interface) is a high-speed, short-distance bus that links a single controller (the STM32 on our glove PCB) to one or more peripherals—in our case the 9-DoF ICM-20948 IMU. The bus uses up to four dedicated lines: a clock (SCLK) from the master, a data-out line (MOSI) for commands, a data-in line (MISO) for sensor replies, and an active-low chip-select (CS) that lets us share the same wires with additional devices if needed. Unlike I²C’s open-drain “wired-AND” scheme, SPI employs push-pull drivers, so each edge is clean and timing is precise; with the STM32 we clock the IMU at several MHz, fast enough to stream gyro and accelerometer data in real time. All transfers are full-duplex: while the MCU shifts out an 8-bit register address on MOSI, the IMU simultaneously shifts its reply onto MISO, keeping latency to a minimum. To bridge the IMU’s 1.8 V logic to the STM32’s 3.3 V domain we insert small MOSFET level-shifters, preserving signal integrity without sacrificing speed. This simple, low-overhead protocol lets the firmware sample orientation data continuously and fuse it with the flex-sensor readings, giving the VR application an accurate, up-to-date snapshot of the hand’s pose on every frame. The expected (successful), golden SPI signal is recorded on the Oscilloscope in the lab, as shown in **Figure 12**.

Figure 12: This is an example of a ‘golden’ SPI transaction as recorded by us in the lab

Software Implementation of the IMU

Unlike the setup for the flex sensors, the set-up for IMU was done by carefully following the set-up instructions as provided in the following github link:
mokhwasomssi/stm32_hal_icm20948: ICM-20948 library with STM32 HAL driver
*code aspects are highlighted in green text

Our firmware uses a lightweight driver pair icm20948.c / .h (≈3 kB) that talks to the sensor over SPI1 at 7 MHz. The logic lives in three layers:

1. **Low-level register access**
   - icm20948_spi_write(reg, val) and icm20948_spi_read(reg, len, *buf) wrap HAL_SPI_TransmitReceive() and automatically toggle CS
   - A 10 µs delay after every multi-byte burst satisfies the sensor's 9-clock turn-around spec.

2. **Device bring-up (icm20948_init)**
   - Resets the chip, waits 100 ms, verifies WHO_AM_I == 0xEA.
   - Enables the internal 20 MHz oscillator, sets gyro ±250 dps, accel ±2 g.
   - Routes the AK09916 magnetometer through the internal I²C master and parks it in continuous-mode 100 Hz.
   - Writes factory calibration biases into the gyro/accel offset registers. Bias values are generated once with a "board flat" script and hard-coded in icm20948_cal.h.

3. **Sampling API**
   - icm20948_gyro_read(&axises) / icm20948_accel_read(&axises) pull six bytes from the respective OUT registers, combine high/low, and scale to mdps/mg.
   - ak09916_mag_read(&axises) reads the STATUS1 + XYZ bytes through the sensor's I²C pass-through.
   - All three calls are non-blocking and finish in < 250 µs each.

Main loop integration

```
icm20948_init();

ak09916_init();

/* 100 Hz polling inside while(1) */

if (tick_1ms % 10 == 0)          // every 10 ms

{

        icm20948_gyro_read(&my_gyro);

   icm20948_accel_read(&my_accel);

   ak09916_mag_read(&my_mag);

   printf("G:%d,%d,%d A:%d,%d,%d M:%d,%d,%d\r\n",
```

```
        my_gyro.x,  my_gyro.y,  my_gyro.z,

        my_accel.x, my_accel.y, my_accel.z,

        my_mag.x,   my_mag.y,   my_mag.z);
}
```

- **Printing** uses the PUTCHAR_PROTOTYPE hook to mirror every character to **LPUART1** (USB-TTL) *and* **USART2** (ESP32 bridge) simultaneously.

Error handling

- Every register write is followed by a read-back check. If any mismatch occurs the driver asserts ICM_FAIL and calls Error_Handler() which blinks **PA7** at 2 Hz and waits for SWD attach.
- If the magnetometer fails to respond, the system keeps running in **6-DoF** mode and sets a mag_valid = 0 flag so the host knows to ignore heading.

Performance

- **Sample/print latency** ≈ 1.6 ms for all nine axes at 100 Hz.
- Combined with the 50 Hz radio pipeline (Section 4.3) this preserves hand-tracking lag below the 60 ms VR budget.

This layered approach keeps the IMU code portable, lets us switch to DMA or burst-FIFO later, and isolates sensor quirks behind a simple three-function API.

Overview of the ESP32 (the wireless communication bridge)

The ESP32 is our glove's wireless messenger. It takes the data the STM32 has already packaged—finger bends from the flex sensors plus motion from the IMU—and sends it over Wi-Fi or Bluetooth straight to the VR computer. The computer can also talk back, letting us tweak settings or update code without plugging in a cable. In short, the ESP32 keeps the glove lightweight and cable-free while still delivering quick, reliable data for smooth hand tracking. For the purpose of this project we used the ESP32 wroom-32-e for its modern and advanced wireless communication technology.

Why use the ESP32 for wireless communication?

We picked the ESP32 because it gives us everything we need for a wireless glove in one cheap, tiny chip: built-in Wi-Fi and Bluetooth for plenty of range, enough bandwidth to stream hand-tracking data with low latency, lots of example code and community support, and low power draw so the battery lasts. Using the ESP32 also means we don't need a separate radio module or complicated RF layout, keeping the PCB simple and costs down while letting us push over-the-air updates or calibration tweaks whenever we want.

However, it is important to acknowledge the challenges related to using an ESP32 MCU in our PCB Design, as these can usually be attributed to requiring 'extra' considerations like a

meticulous antenna layout and possible noise. However, considering the benefits clearly outweigh the cons in our case, we took extra care while designing the ESP32 subcircuit, taking extra care to ensure that this issues were duly mitigated

Hardware Implementation of the ESP32

The ESP32-WROOM module (U2) is powered from the board's 3.3 V rail and referenced to the surrounding ground plane through its thermal pad. Its enable pin, EN/CHIP_PU, must be held high for the radio to run, so a 10 kΩ pull-up (R1) ties it to 3.3 V. A reset push-button (SW1) shorts EN to ground through a small 0.1 µF capacitor (C4); the cap adds a brief RC delay that filters switch bounce and guarantees a clean low pulse whenever you press the button. UART0 pins (U0TXD / GPIO1 and U0RXD / GPIO3) are broken out as TX_ESP32 and RX_ESP32 so you can plug in a USB-to-serial adapter for flashing or debugging, but they're left idle during normal glove operation (R17 provides a mild 1 kΩ series resistance to protect the line if someone hot-plugs a cable).

Note: GPIO0 must sit high at power-up for normal code execution; pulling it low while toggling EN drops the ESP32 into its UART bootloader so you can flash new firmware. Relying solely on the module's internal pull-up is fragile—noise can momentarily drag the pin low and strand the chip in bootloader mode—so add an external 10 kΩ pull-up (plus an optional test-point you can short to ground when intentional re-flashing is needed).This is one key revision we had to make to the final board, and we would redesign this part accordingly in order to prevent unnecessary wired external connections to the PCB.

Software Implementation of the ESP32 subcircuit

The implementation of the ESP32 as a wireless bridge was a relatively simple task as (a) it was easier to set up the environment in comparison to the STM32 (b) having worked with the wireless capabilities of ESP32 on previous personal projects, our team members were very well versed in building the necessary components that constitute of the wireless bridge. However, with that being said, we still had to deal with some last minute fixes and recurring latency issues, we will talk about these aspects in detail in the subsequent sections.

Wireless Data Aggregation & Transmission (STM32 → ESP32 → PC)

Role	Device	Link	Data Rate	Purpose
Sensor hub	STM32L031K6T7	UART → ESP32	921 600 baud	Packages three flex-sensor readings every 10 ms
Radio bridge + IMU reader	ESP32-WROOM-E	ESP-NOW → PC	≈ 25 kB s^{-1}	Merges flex data + IMU data, transmits wirelessly

Why ESP-NOW?

- Ultra-low overhead – 1–5 ms MAC-layer frames, no TCP/IP handshake.
- Zero-config pairing – Just a 6-byte MAC exchange; works on any desk.
- Range vs. power – 20 m indoor link at 8 dBm keeps the Li-ion cell alive all afternoon.

Last-Minute IMU Work-around

Problem: A hairline break under the on-board level-shifter severed the SPI path between the IMU and STM32.
Fix: we wired a SparkFun ICM-20948 breakout straight to spare ESP32 pins (MOSI 18, MISO 19, SCLK 5, CS 23, plus local 1 µF decoupling).
Effect: The ESP32 now polls the IMU at 50 Hz, merges its quaternion with the UART flex frame, and ships a 28-byte payload via ESP-NOW—still meeting the glove's ≤ 50 ms latency budget.

Throughput & Stability Tuning

- Rate alignment – IMU throttled to 50 Hz; STM32 ADC loop matched.
- DMA UART – ESP-IDF driver switched to DMA, cutting ISR load ≈ 65 %.
- Framed packets – 32-bit timestamp
- Staggered bursts – Offsetting IMU and flex frames keeps bursts < 2 kB s^{-1}.

Packet Format (rev A)

```
uint32_t timestamp;
int16_t  flex0, flex1, flex2;
uint8_t  dataReady;
float    accX, accY, accZ;
float    gyrX, gyrY, gyrZ;
```

Outcome

< 0.1 % packet loss and 50–65 ms lag from hardware to the software receiver.

Computer-based Programs (Data Bridge + Unity VR)

PC Bridge (Python)

- The script main.py runs on the host PC and performs three jobs in real time:
- Serial ingest – Opens COM4 at 115 200 baud and continuously reads ASCII lines of the form
- Flex0: 2150 Flex1: 2040 Flex2: 1980 produced by the glove's ESP32.
- Flex-to-angle mapping – Converts raw ADC counts to joint angles using empirically tuned ranges (Flex0 → Index, Flex1 → Middle, Flex2 → Ring). Thumb and Pinky angles are approximated from the Middle finger to keep the hand looking natural when only three sensors are available.
- TCP forward to Unity – Maintains a persistent socket to 127.0.0.1:5065. Every time a complete frame is parsed it sends one JSON line per finger:

```
{"hand":"Left","finger":"Index","angle":23.4}
```

The transmission rate is limited by the serial loop; at 115 200 baud and 10-bit ADC values the pipe comfortably sustains 100 packets s^{-1} with < 5 ms jitter.

Here are some key implementation points we noted:

- Robust Unity auto-reconnect: the script retries the socket every 5 s until Unity starts.
- 2) Empirical calibration block (map_flex_to_angle) lets you paste new min/max counts without touching Unity.
- Ring / Pinky synthesis: Ring = 0.95 × Middle, Pinky = 1.10 × Middle—gives a believable curl when gripping.

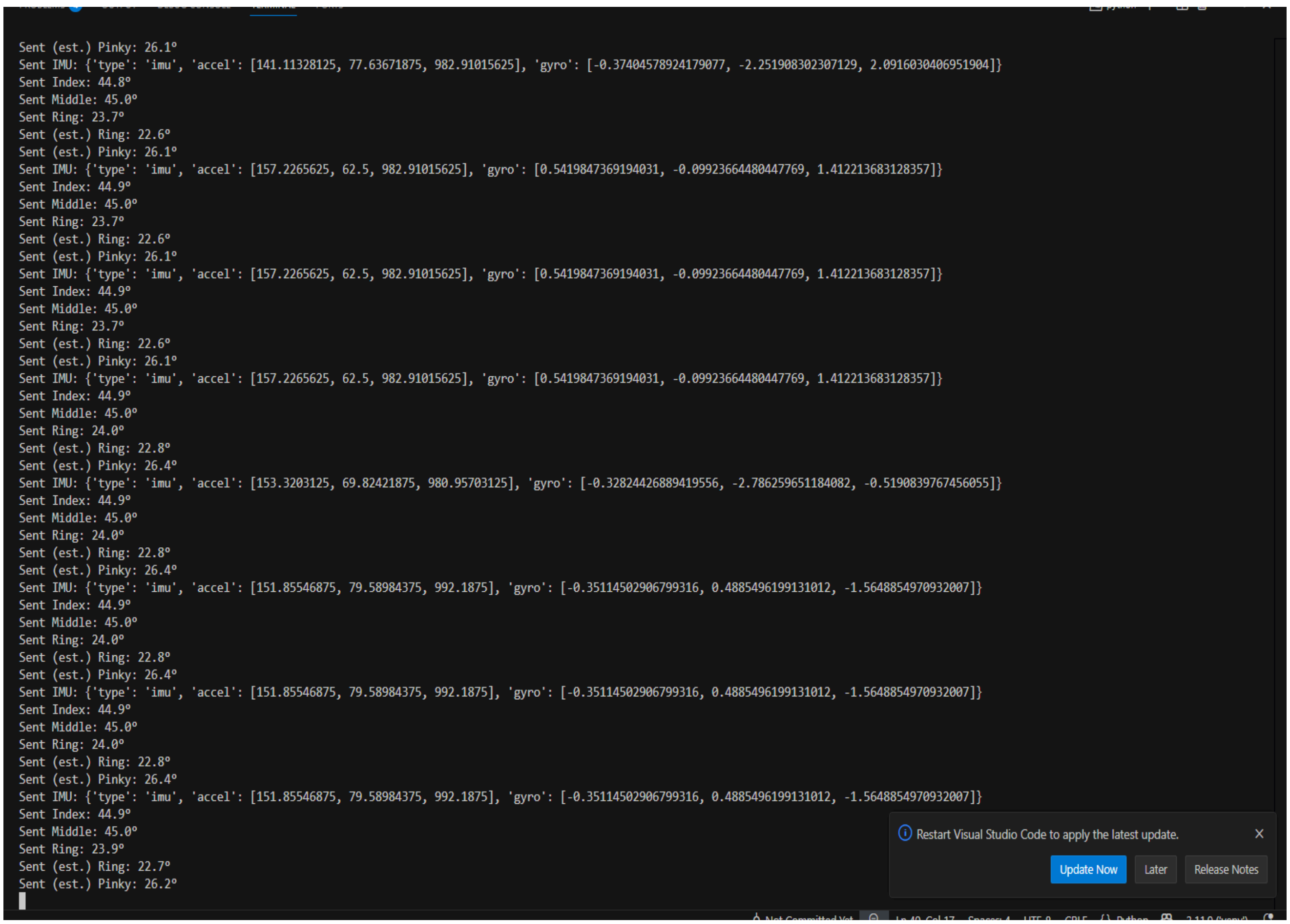


Figure 13: main.py output - output of serial data from the receiver module

Unity Listener (C#)

A lightweight C# script (GloveReceiver.cs) runs on an empty GameObject in the scene. This script performs the following tasks:

1. Opens a TcpListener on port 5065
2. Reads each newline-terminated JSON message on a background thread
3. Deserialises to a GlovePacket struct { string hand; string finger; float angle; }
4. Looks up the corresponding Transform (or Humanoid bone) and sets
   localRotation = Quaternion.Euler(angle, 0, 0);.

```
void UpdateFinger(string finger, float deg)
{
    Transform t = fingerMap[finger];          // e.g. "Index" → leftHandIndex1
    Quaternion target = Quaternion.Euler(deg, 0f, 0f);
    t.localRotation = Quaternion.Slerp(t.localRotation, target, 12f * Time.deltaTime);
}
```

Using Quaternion.Slerp with a high interpolation speed smooths out micro-jitter yet keeps total latency under one rendered frame (≈ 16 ms at 60 Hz).

End-to-end Timing

- Serial read → angle mapping ≈ 1 ms
- TCP send/recv ≈ 0.5 ms (loopback)
- Unity JSON parse + bone update ≈ 1.5 ms

Total software overhead ≈ 3 ms, so the overall hand-tracking latency is dominated by the 45–50 ms radio path as described in the previous section

Future Improvements to Computer Based Programmes

- Switch the Python bridge to pyserial-asyncio and uvloop to free the main thread for logging/GUI.
- Add a small on-screen calibration panel in Unity to live-tune flex_min / flex_max.
- Migrate to the new Unity XR Hands package and drive Meta Quest hand bones directly, eliminating custom finger maps.

With this pipeline the glove streams sensor data straight into Unity in under 3 ms of PC overhead, giving an end-to-end feel that is responsive enough for simple VR manipulation tasks and demos.

Additional PCB Components

One of the most critical components of our PCB Design involves the routing of different kinds of power to different chips/components involved in the board. Since we do not want to use multiple voltage inputs to the board at a single time, we make use of LDOs to convert the standard 5V supplied to the board.

In **Figure 14**, Two low-dropout regulators create the rails the glove needs: the **TC1262-33** drops the incoming 5 V supply to a clean **3.3 V** for the STM32, ESP32, flex-sensor buffer, and logic level-shifters, while the **AP2112K-1.8** taps that 3.3 V line and generates a precise **1.8 V** rail for the ICM-20948 IMU's core and I/O. Local 1 µF ceramics (C10, C13, C19) sit next to each regulator's input and output pins to keep both rails stiff during load transients. Providing dedicated, well-regulated voltages prevents over-stressing the sensor, guarantees full logic-level swing for the MCUs, and keeps analog readings free from power-supply noise—all of which are essential for accurate, low-latency hand tracking.

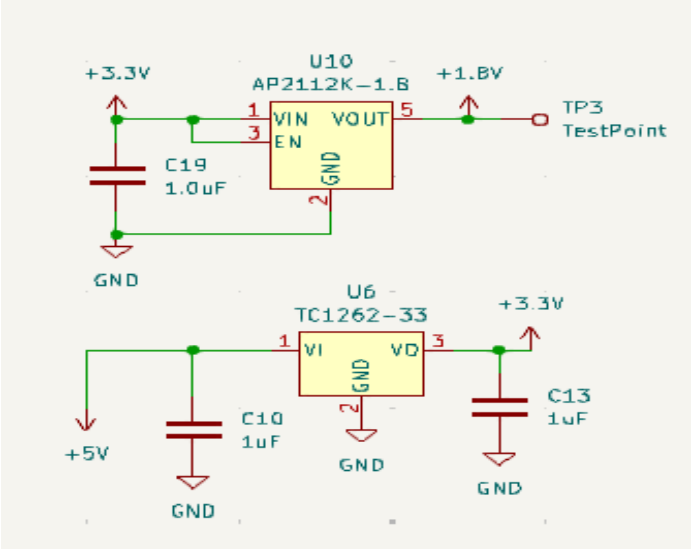


Figure 14: Image showing the schematics of the two LDOs used

A simple power or status LED is one of the cheapest, most effective diagnostic tools on any board. It gives an immediate visual cue that a rail is up, helps spot shorts or blown fuses without a multimeter, and can double as a user-feedback channel (e.g., blinking for pairing or error codes). Because LEDs draw only a few milliamps and demand no firmware on day one, they provide "instant reassurance" during bring-up and field servicing.

In Figure 15, a 1 kΩ resistor lets a few-milliamp current flow through the LED whenever the 3.3 V rail is alive, giving an immediate visual "power-on" confirmation. That single dot of light is the

quickest way to spot a blown fuse, dead regulator, or accidental shutdown—no multimeter or serial console needed.

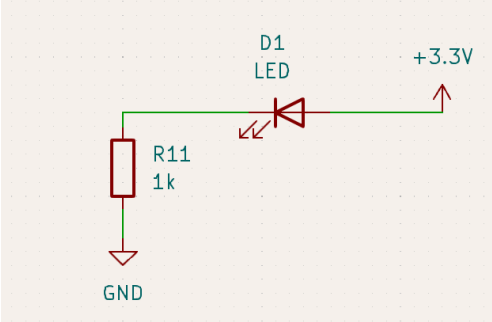


Figure 15: Image showing the 3.3V indicator LED we drew on the PCB

PCB Design Tips

During our journey creating this project, we learnt a great deal about general PCB Design Practices, and these are some of the practices that we felt we most positively affected our project:

- Maintain a continuous ground plane. Allocate an entire layer—or at minimum the bottom copper—as a solid GND pour and stitch it liberally with vias to reduce EMI, provide low-impedance return paths, and simplify routing. We did exactly this for one of the copper layers in our PCB
- Place decoupling capacitors at every supply pin. Position 100 nF ceramics within 2–3 mm of each IC's VDD/VCC leads and add bulk capacitance (≥ 1 µF) at each rail entry.
- Respect RF keep-out zones. This means remove all copper, vias, and silkscreen beneath on-board antennas and observe the manufacturer's keep-out distance (typically 15 mm for 2.4 GHz chip antennas). This is seen in the case of the ESP32 microcontroller as the current copper pour extends into the antenna keep-out at the top-left corner; retract the plane by at least 15 mm and apply a hatched keep-out in KiCad to preserve RF efficiency.
- Matched-length SPI routing. Although specific to our case, it is imperative to keep CS, SCLK, SDI, and SDO traces **equal** length and away from the antenna feed to minimize skew and crosstalk—we used KiCad's interactive length-tuning (Ctrl + L) to do this.
- Prefer 45 ° trace bends. Acute angles mitigate acid traps during etching and reduce impedance discontinuities on fast edges.
- Run ERC/DRC regularly. Perform electrical and design-rule checks at every major routing milestone to catch clearances, unconnected pins, and via anomalies early.

General Project tips

- Having basically lived in the ECE395 ADSL lab for these last few weeks, we felt it was important to share some quick project survival tips: some based on wisdom passed down by our seniors while some others we unfortunately learned the hard way. They are as follows:
- Version-control everything. Put schematics, firmware, and even board files in Git so you can roll back that “quick tweak” at 2 a.m.
- Blink first. Before complex code, make an LED flash to confirm power, clock, and programming chain are solid.
- Prototype on headers. Add a row of spare GPIO/SPI pads—this is especially useful in case you want to test if there's a hardware issue with your on-board components
- Keep a living BOM. Track part numbers, suppliers, and lead times in one sheet; shortages hurt more than schematic errors.
- Test in layers. Verify power, then comms, then sensors—isolating faults is easier than untangling a full system crash.

Access to Codebase

We have use git to version control this project, If you would like to access the codebase for this project please use the following link: https://github.com/archi-max/GloveControlUnit/tree/main

Conclusion

In the end, our wireless-glove project delivered a fully functional prototype that fuses flex-sensor data with IMU orientation and streams it to a host PC over Wi-Fi—exactly the experience we set out to create. The design combined a quad-op-amp buffer, dual-rail power architecture, an STM32 sensor hub, and an ESP32 radio on a compact two-layer PCB, all validated through careful decoupling, level-shifting, and antenna layout. Getting there was not without hurdles: shipment delays, an overlooked external pull-up on GPIO0 occasionally stranded the module in bootloader mode, noisy unplated vias forced last-minute jumper wires, meant that we were constantly working till the hours before the demo. Despite these challenges—and the inevitable late-night rework sessions—system bring-up succeeded to a large extent: the board powered on cleanly, sensors calibrated, packets flowed wirelessly, and the VR scene mirrored every finger bend and palm movement in real time. However, we faced some issues with cold solder joint that caused the transmission program to stop working after a period of time, but these issues can be fixed with more meticulous and professional PCB layout and soldering practices we listed in the previous section. Despite this particular issue, we believe that wireless glove control units can be expanded to a myriad of applications such as robotics, computer vision etc., and will surely pursue a different application with similar fundamentals given the opportunity to take this class again.