HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

FINAL PROJECT

EMBEDDED SYSTEMS

COURSE: IT4210E — CLASS CODE: 156819 — SEMESTER 2024.2

Le Ngoc Quang Hung	20225975	hung.lnq225975@sis.hust.edu.vn
Tran Quang Hung	20226045	tuan.tp22h6045@sis.hust.edu.vn
Tuong Phi Tuan	20226069	tuan.tp226069@sis.hust.edu.vn

Submitted to: Professor Ngo Lam Trung

Hanoi - 2025

HE16: Hall-Effect Keyboard

Glossary

Term	Definition
Hall-Effect sensor	Sensors that detect magnetic fields and convert them into electrical signals.
Hall-Effect switch	A type of contactless switch that uses a magnet and a Hall effect sensor to detect key presses based on the magnetic field strength.
Actuation point	The depth at which a keypress is registered/unregistered and a signal is sent to the computer. In mechanical switches, this is the fixed point where metal contacts close the circuit. In Hall effect switches, it can be adjusted via firmware.
HID(Human Interface Device)	A class of devices that interact with humans, like keyboards, mice. In USB, it defines a protocol for such interactions.
CDC(Communications Device Class)	A USB class to emulate serial ports, allowing microcontrollers to communicate via a virtual COM port.
Rapid Trigger	A feature allowing keys to reset and re-register quickly, enabling faster repeated input without returning to the original actuation point.
Macro	A sequence of key presses/actions recorded and replayed with a single key.
EWMA (Exponentially Weighted Moving Average)	A smoothing algorithm that weights recent data points more heavily to reduce noise.
USB descriptor	Structured USB data describing the device's capabilities (e.g., interfaces, endpoints) used during enumeration.

1. Introduction

1.1. Project Description

The goal of this project is to recreate a basic version of an analog Hall-Effect keypad, offering a wider range of functionality compared to traditional keypads.

Traditional keypads commonly rely on a keyboard matrix to detect button presses, where each key is either in an ON (HIGH) or OFF (LOW) state. These inputs are then translated to HID signals that a computer can interpret.

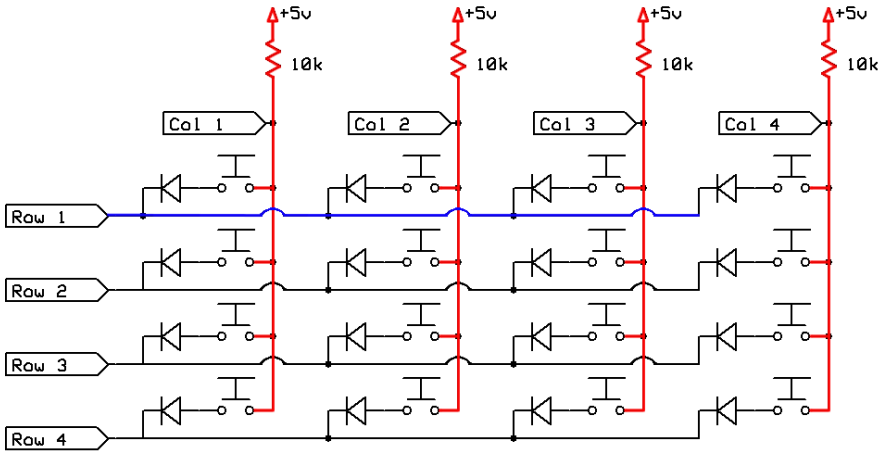


Figure 1: A keyboard matrix

In contrast, this project replaces the standard mechanical switches with Hall-Effect sensors and magnetic key switches. Unlike typical digital switches that only detect binary states (pressed or not pressed), Hall-Effect sensors provide a continuous analog voltage corresponding to the distance between a magnet and sensor which in this case, is how far the key is pressed.

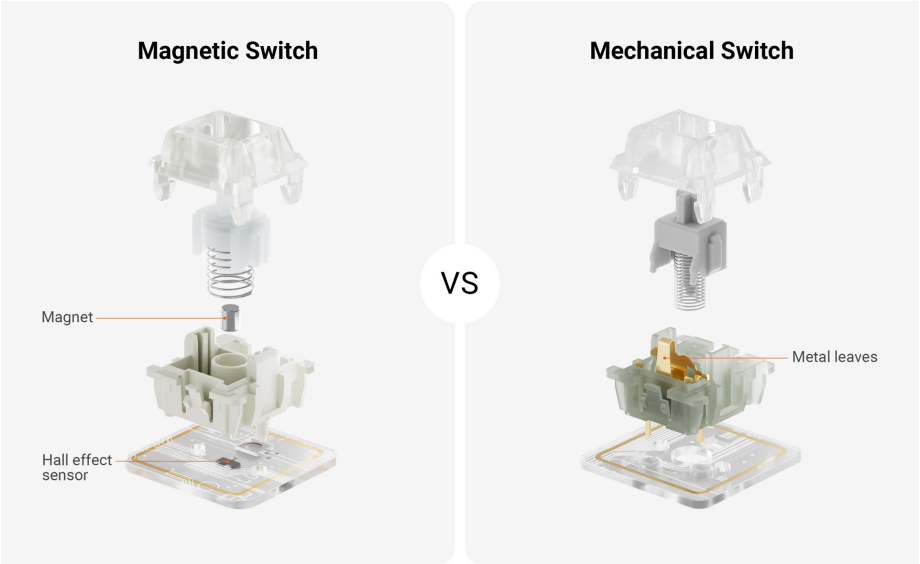


Figure 2: Structure of a magnetic switch and a mechanical switch

Since the Hall sensor outputs an analog signal, it enables a finer level of input control. This makes it ideal for tasks requiring variable input, such as joystick movement, pressure-sensitive key presses, or velocity-based control in digital instruments and gaming environments.

Moreover, since there are fewer mechanical parts (no metal leaves and output pins) in magnetic switches, there are fewer points of failure, helping with the durability of the board.

1.2. Project Requirements

1.2.1. Functional Requirements

The keypad supports the following functionalities:

- Detect analog signals from the Hall-Effect sensors based on how deep the switches are pressed and translate them into HID inputs.
- Support rapid trigger, which is a feature often found in analog gaming keyboards that uses analog sensing to detect key positions continuously. Instead of waiting for keys to fully reset before registering another press, they can start registering again as soon as they start moving upward. This allows for much faster repeated input than traditional mechanical switches.
- Support macro functions, allowing users to assign sequences of keystrokes to a single key for automated execution of complex actions.

- Operate as a composite USB device with the following interfaces:
  - A HID interface for key input.
  - A CDC interface for configuration and debugging.
- Provide a debug and configuration interface over a serial protocol (via the CDC interface), eliminating the need to recompile the firmware for changes.
- Provide an auto calibration method to compensate for inherent signal differences between individual sensors, ensuring consistent key behavior across the keypad.

1.2.2. Non-Functional Requirements

- Include appropriate analog filtering (e.g., low-pass filter and software filter) to stabilize sensor outputs. For hardware filter, voltage fluctuations must fall within ±10 mV. .
- Support a polling rate of 1 kHz, the keyboard must spend at most 1 tick (1ms) to complete a loop.

2. Design

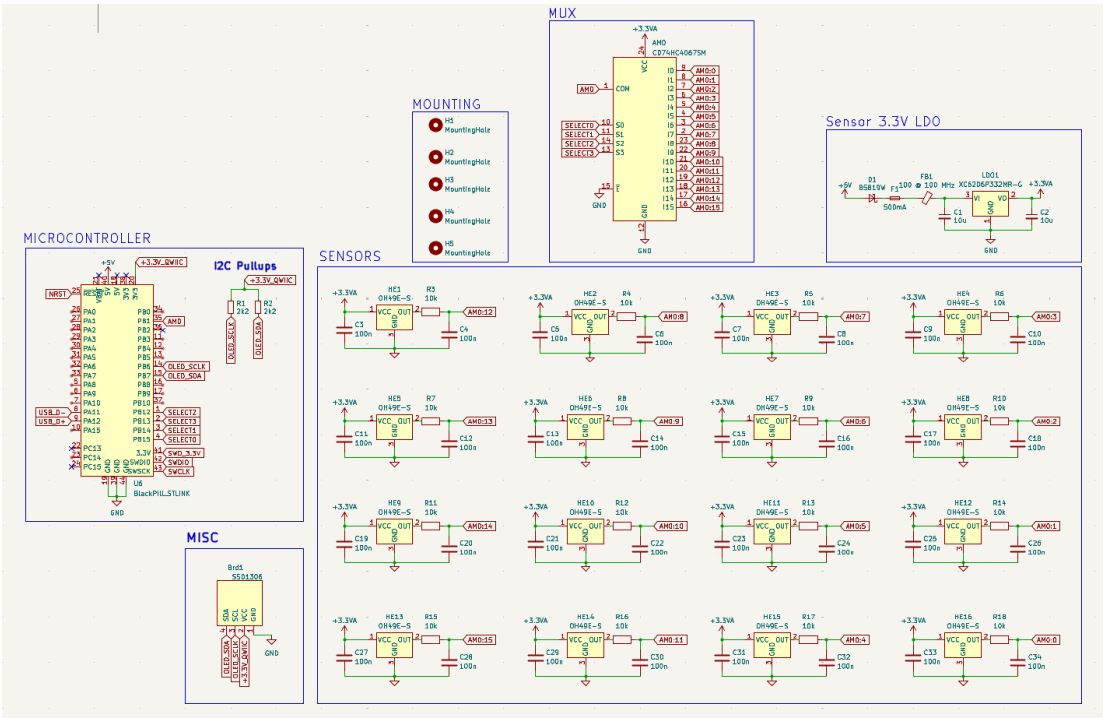
2.1. Function Assignments

Function	Hardware	Software
Detect Analog Signals and Translate them into HID Inputs	Yes	Yes
Rapid Trigger	No	Yes
Macro	No	Yes
Composite USB Device	No	Yes
Debugging interface	No	Yes
Analog filters	Yes	Yes
1 kHz polling rate	No	Yes
Calibration method	No	Yes

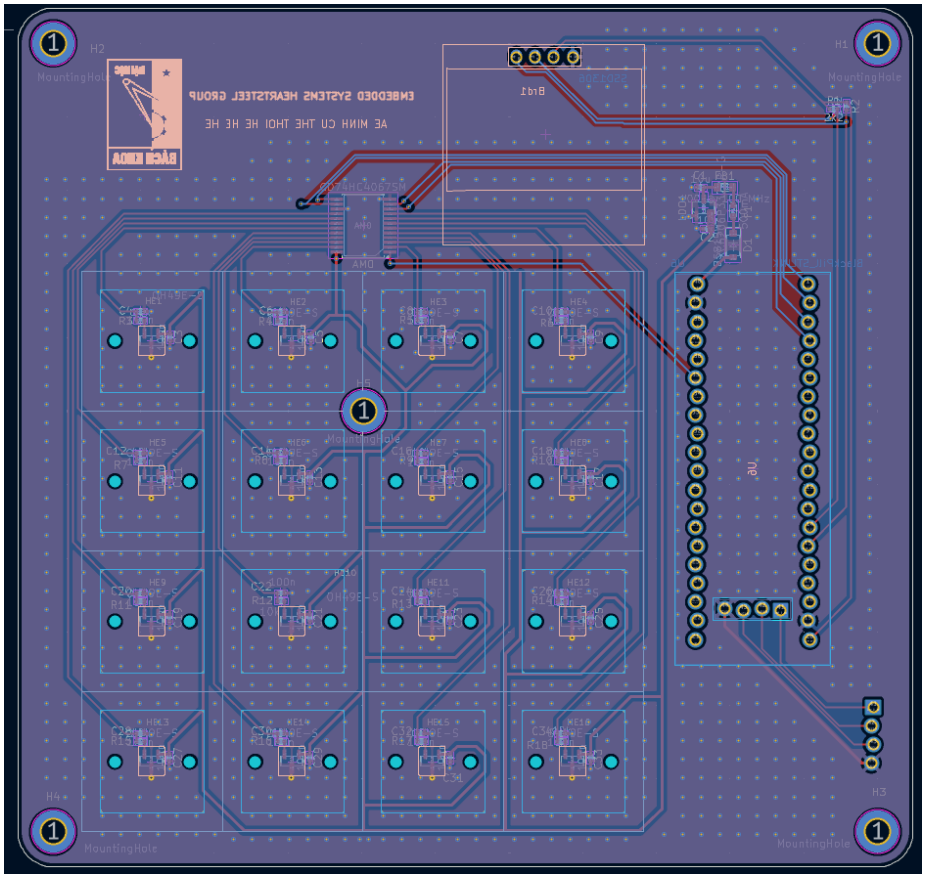
2.2. Hardware Design

For the hardware,a PCB is used instead of a breadboard to minimize electrical noise that could interfere with the sensors. Additionally, some components are only available in surface-mount (SMT) packages, which are better suited for PCB integration.

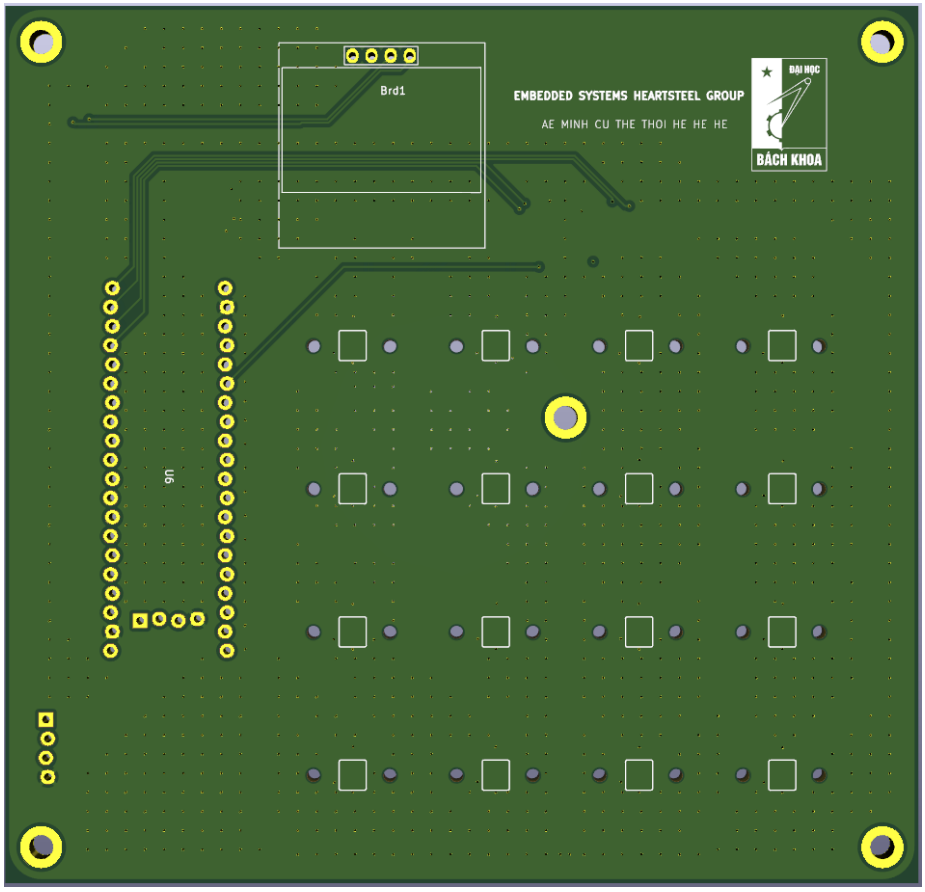
System Schematic



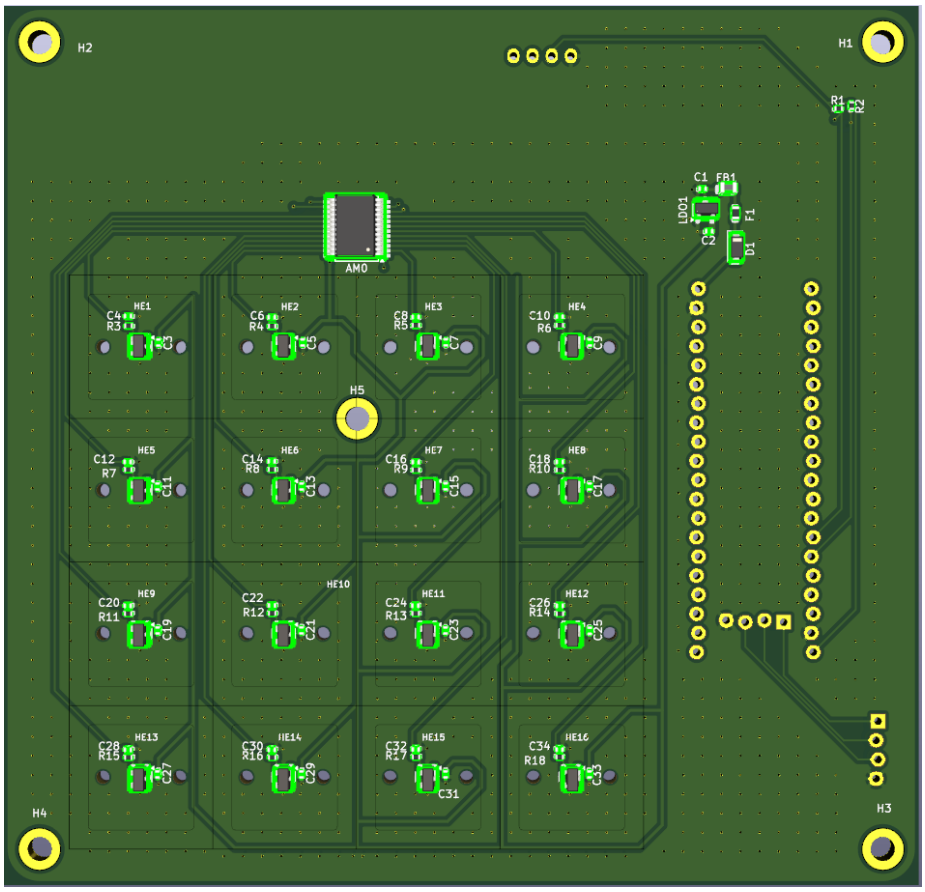
PCB Design



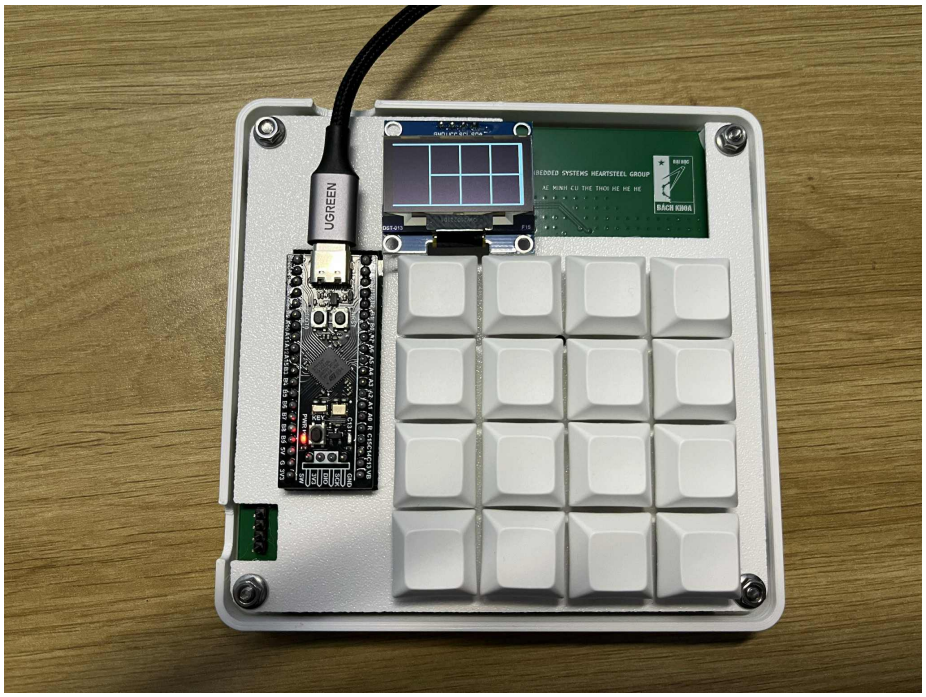
3D View Front



3D View Back


H2
H1
H5
H4
H3
AM0
LDO1
C1
FB1
F1
D1
R1
C2
HE1
HE2
HE3
HE4
HE5
HE6
HE7
HE8
HE9
HE10
HE11
HE12
HE13
HE14
HE15
HE16
C31

Final Product



Components

Component	Description	Datasheet
STM32F411CEU6 WeAct Blackpill Kit	A compact development board based on the STM32F411CEU6 ARM Cortex-M4 microcontroller.	ST Datasheet
SSD1306 OLED Display	Standard 128x64 OLED display.	SSD1306 Datasheet
XC6206P332MR-G	A 3.3V low dropout voltage regulator used to step down and regulate power supply. Provide a stable analog power source for the sensors.	XC6206 Datasheet
CD74HC4067SM96	A 16-channel analog multiplexer/demultiplexer. Used to expand the number of analog inputs to the microcontroller, allowing multiple Hall effect sensors to be read using fewer ADC pins. (On the board, all 16 analog sensors can multiplex into only one ADC pin)	CD74HC4067 Datasheet
OH49E-S	A linear Hall-Effect sensor that outputs an analog voltage proportional to the magnetic field strength. Used to detect the position of a magnet in magnetic switches.	OH49E Datasheet
Gateron KS-20	A magnetic key switch designed to be used with Hall effect sensors.	Gateron Datasheet

Microcontroller

The STM32F411CEU6 microcontroller is chosen mostly for its low price and decent performance. It also supports I2C and USB 2.0 full-speed protocols, which is enough for the scope of this project.

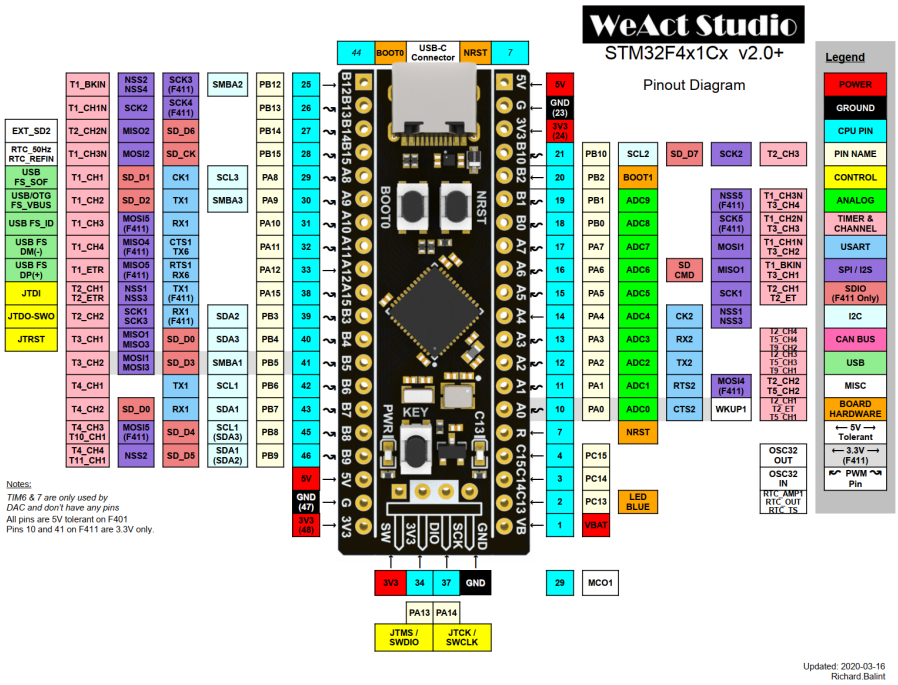


Figure 3: WeAct STM32F411CEU6 Blackpill

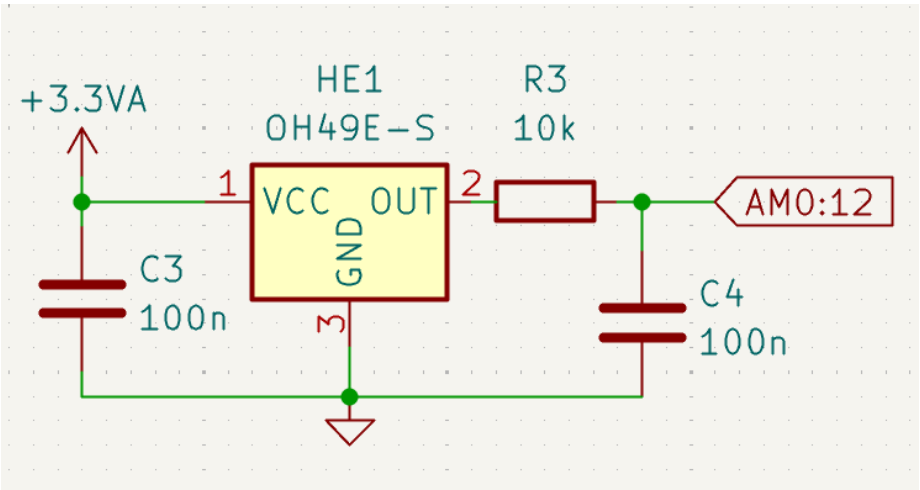
Two pull-up resistors are also connected to the I2C pins to pull them high since I2C devices can only drive the line low and cannot pull the line up themselves.

Power Supply

To minimize noise and ensure reliable analog signal, the system uses two separate 3.3V power sources for digital and analog domains:

- **3.3V_QWIIC**: This is the default 3.3V supply provided by the development board (e.g., STM32 Blackpill). It powers digital components such as the OLED display and other logic-level devices.
- **3.3VA**: This is a clean 3.3V analog supply generated by the LDO regulator (e.g., XC6206P332MR-G). It provides a more stable and noise-isolated voltage, used only for analog-related components, such as the Hall effect sensors and the analog multiplexer (e.g., CD74HC4067).

Sensor



For each sensor, we connect a decoupling capacitor to the VCC pin to stabilize the power supply and reduce voltage ripples. On the output pin, we add an RC low-pass filter to suppress high-frequency noise.

The frequency of the changes in the output of the sensor depends on human input, which is not very fast. Following the RC filter formula, the cutoff frequency of 159Hz (R is 10kOhm and C is 100nF) is chosen, which can still comfortably let human input pass, but other high frequency signals will be heavily attenuated.

$$f_{cutoff} = \frac{1}{2\pi RC}$$

Multiplexer

The Blackpill only has 10 analog pins while the keypad has 16 keys. A multiplexer is needed to read all 16 sensors.

The CD74HC4067 is chosen because it is cheap, easy to find, has exactly 16 channels, and there is a DIP version of the module to experiment with before the PCB is ordered.

There are 4 select lines corresponding to 4 selection bits on this module which can be used to select the appropriate channel. For example, if all lines are in LOW state then the 4-bit value is 0000 -¿ channel 0 is selected.

Figure 4: DIP version of the CD74HC4067.

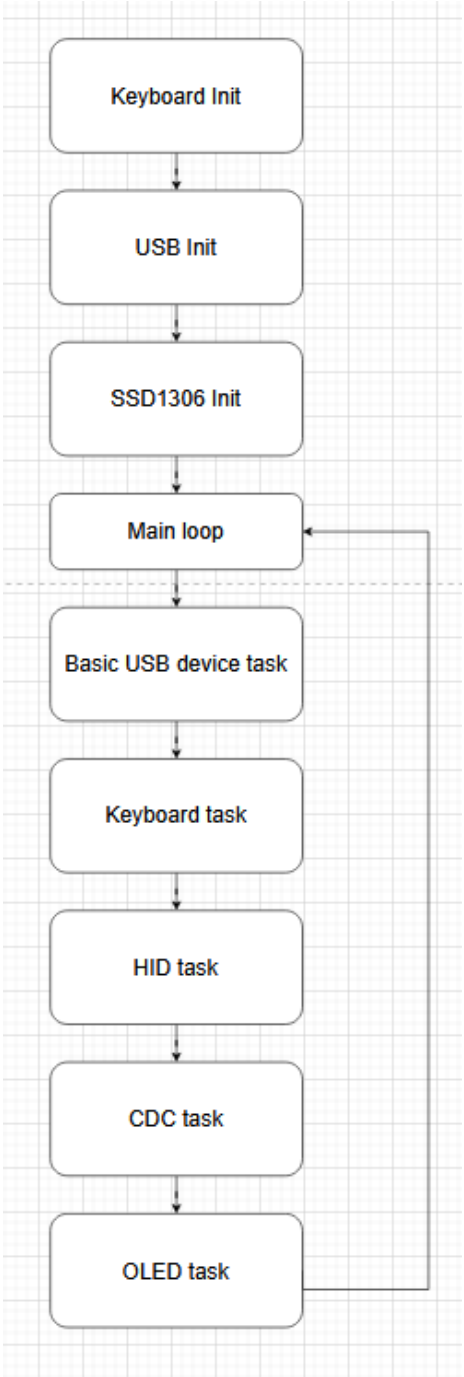
Miscellaneous

An extra I2C 128x64 OLED display and SWD debug pins are also added.

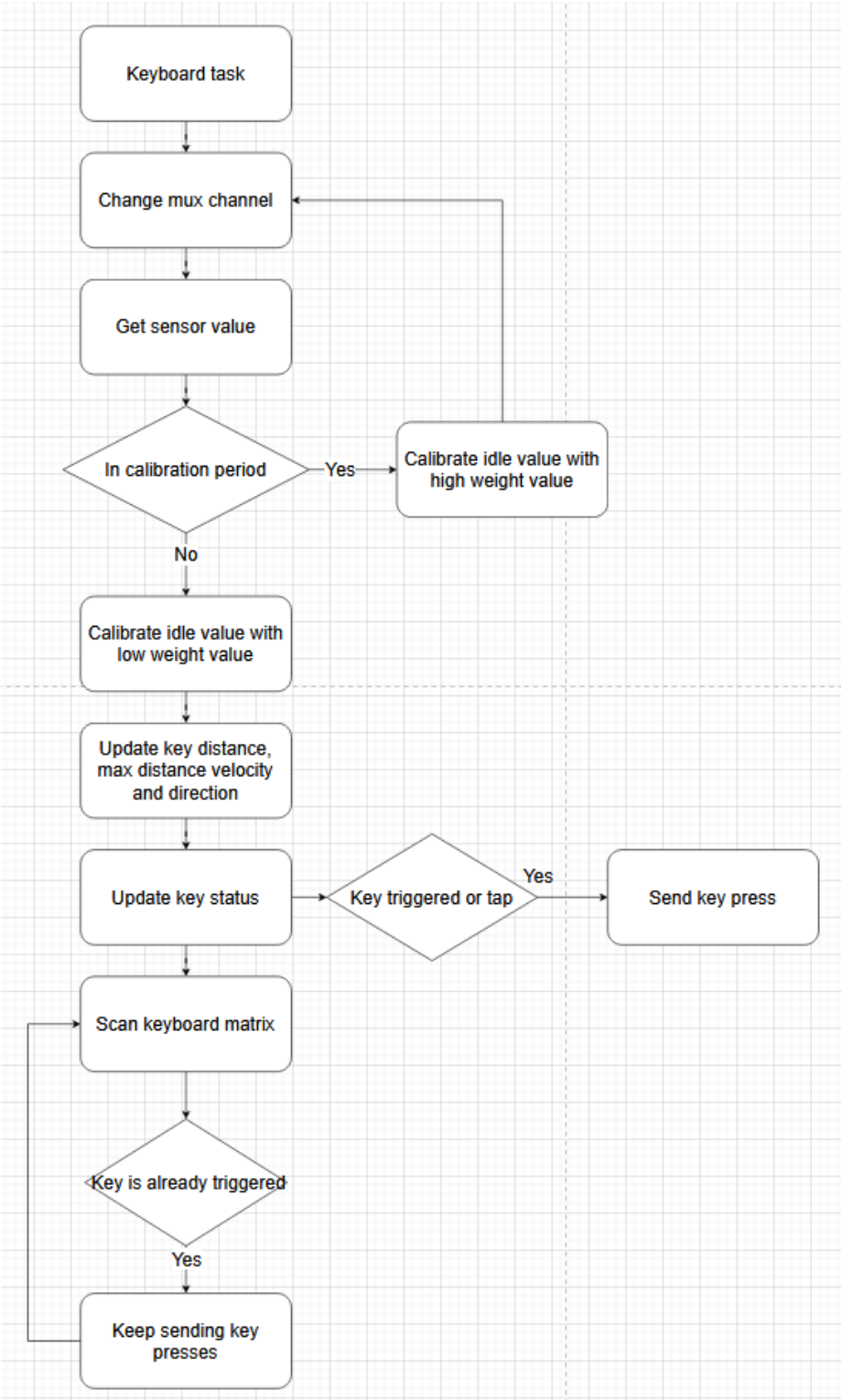
Module	Description
config	Hold the default configurations for the keyboard such as number of multiplexers, number of channels, actuation point offset and more.
main	The main program entry point, responsible for hardware initialization (GPIO, ADC, I2C, USB), and calling the main keyboard processing loop.
keyboard	Implements the core keyboard logic, including managing key states, detecting and processing key presses.
USB	Handling communication through USB using TinyUSB stack.
HID	Submodule of USB, manages the USB HID (Human Interface Device) communication for the keyboard.
CDC	Submodule of USB, manages the USB CDC (Communication Device Class) communication, mainly used to config and debug the board.
OLED	Manage SSD1306 display, handles drawing to a buffer and updates screen.

2.4. System Flow Chart

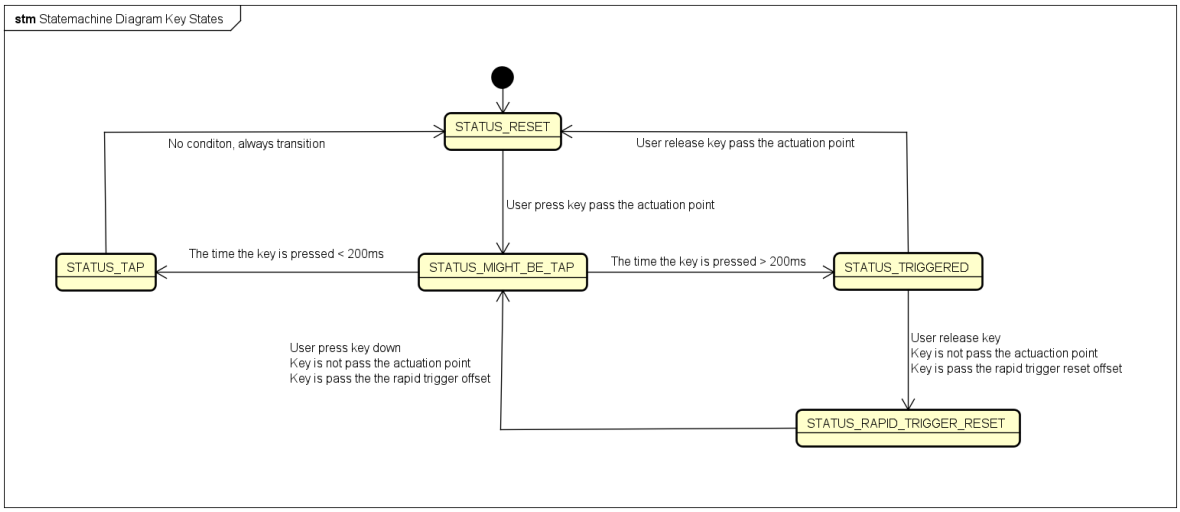
Main Loop



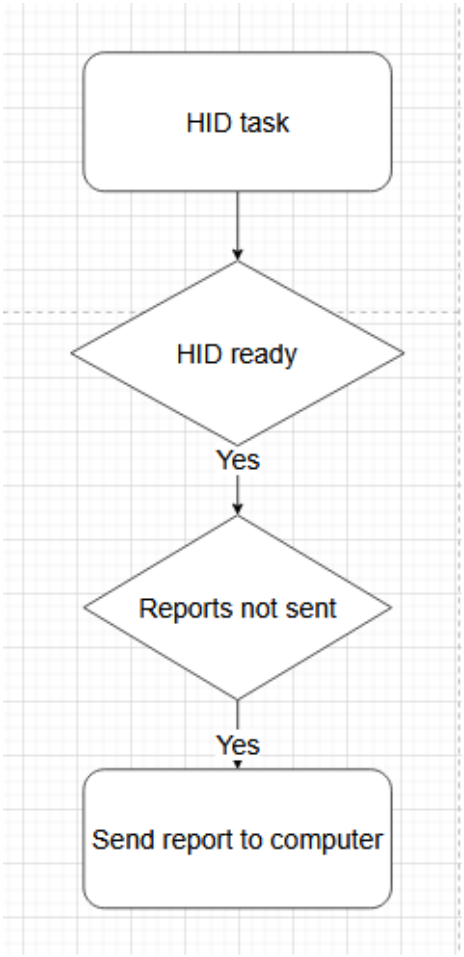
Keyboard Task



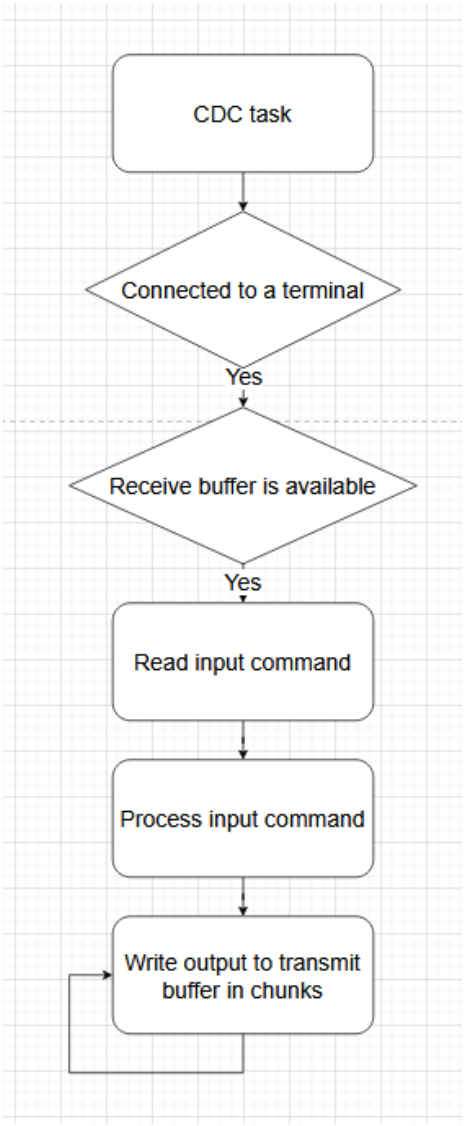
State machine diagram for key status



HID Task

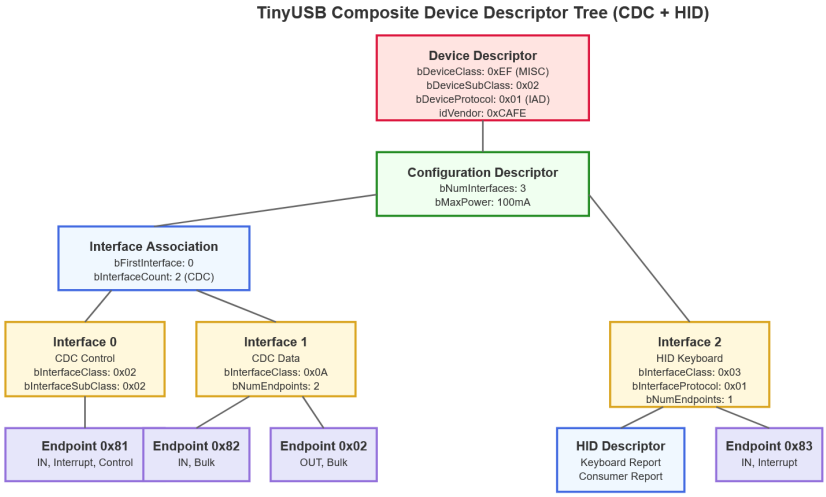

HID task
HID ready
Yes
Reports not sent
Yes
Send report to computer

CDC Task



Even though STM32CubeIDE has its own implementation of a USB stack, it does not support building USB composite devices (HID + CDC). Instead, TinyUSB stack is used.

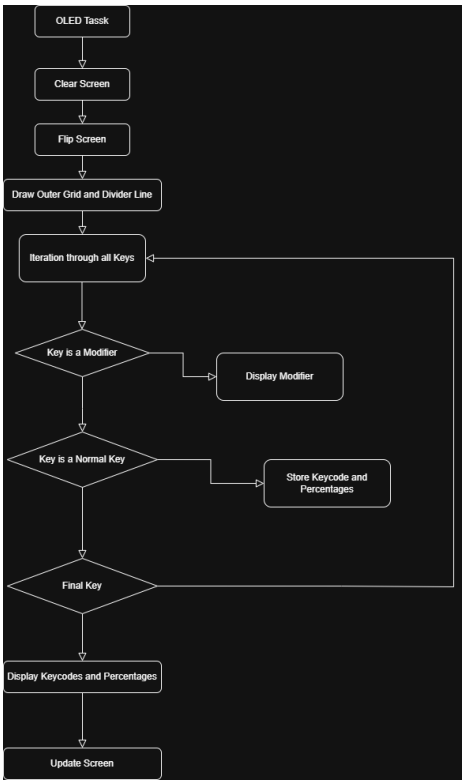
In order to get a composite USB device to work, appropriate descriptors are needed. The image below is an overview of the project's USB descriptors.



There are 3 interfaces:

- **CDC Control interface**: This interface handles the control and configuration of the USB CDC device. It is used for tasks like setting line encoding (baud rate, parity, stop bits), controlling the line state (e.g., DTR, RTS), and managing class-specific requests from the host.
- **CDC Data**: This interface handles the actual data transmission between the USB host and the device for the CDC class. It consists of separate endpoints for IN (device-to-host) and OUT (host-to-device) bulk data transfer.
- **HID interface**: This interface is used for Human Interface Devices like keyboards, mice, and game controllers. It includes a report descriptor that defines the format of the data sent to and from the device. Communication typically happens through interrupt endpoints for low-latency, small-packet data exchange.

OLED Task



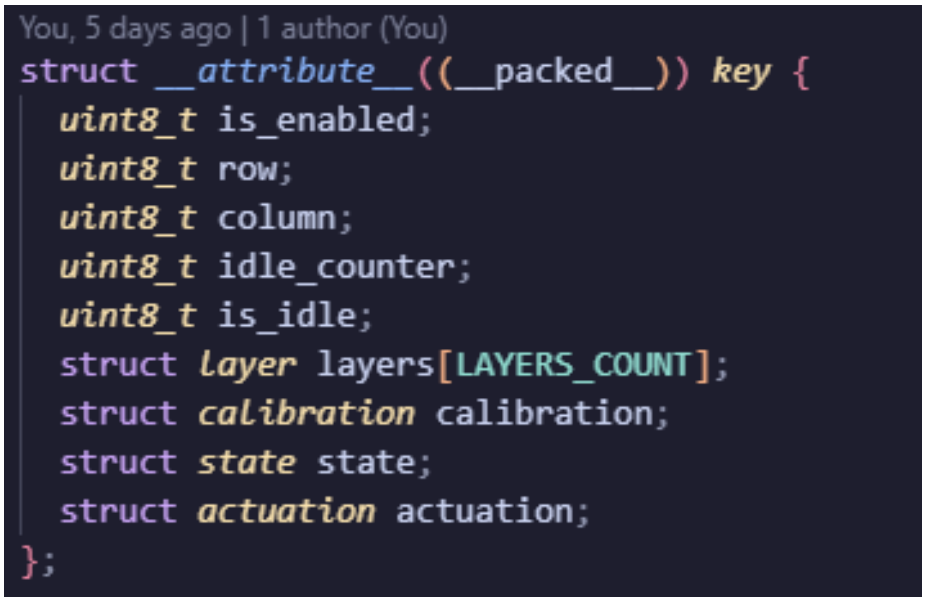
3. Setup and Implementation

The complete source code have been uploaded to GitHub: Embedded System

3.1. Keyboard

The Key data structure is used to store data about a key, including:

- **is_enabled:** the key activation status, disabled keys will be skipped during scanning.
- **row, column:** the position of the key in the keyboard matrix, used to assign keymaps and appropriate channels on the multiplexer.
- **is_idle:** indicates whether the key is in the idle (not pressed) state.



Keymap Layers

A key can have multiple keymaps on multiple layers. For each layer, a key has:

- **type:** The type of key, which can be:
  - `KEY_TYPE_EMPTY`: Default state, mapped to nothing.
  - `KEY_TYPE_NORMAL`: Normal keys (e.g., A, B, 1, 2, 3).
  - `KEY_TYPE_MODIFIER`: Modifier keys (e.g., Shift, Ctrl, Alt).
  - `KEY_TYPE_CONSUMER_CONTROL`: OS-specific consumer control keys (e.g., VOLUME_UP, VOLUME_DOWN on Windows).
  - `KEY_TYPE_MACRO`: Macro keys containing a list of keycodes (e.g., Ctrl+Shift+P).
- **value:** An array of hexadecimal values representing keycodes (e.g., `0x04` for 'A').

```
You, yesterday | 1 author (You)
struct __attribute__((__packed__)) layer {
  enum key_type type;
  uint16_t value[MAX_MACRO_LEN];
};
```

```
enum key_type {
    KEY_TYPE_EMPTY,
    KEY_TYPE_NORMAL,
    KEY_TYPE_MODIFIER,
    KEY_TYPE_CONSUMER_CONTROL,
    KEY_TYPE_MACRO,
};
```

Calibration Data

Used to store data for calibration purposes:

- **cycles_count:** Total number of calibration cycles.
- **idle_value:** The key value when not pressed; recalibrated continuously. Used as the reference for calculating distance.
- **max_distance:** Maximum distance recorded relative to idle_value. Represents the sensor value difference from idle to fully pressed.

```
You, 5 days ago | 1 author (You)
struct __attribute__((__packed__)) calibration {
  uint16_t cycles_count;
  uint16_t idle_value;
  uint16_t max_distance;
};
```

Key State

Stores the current state of the key:

- **value:** Current sensor value.
- **distance:** Difference from `idle_value` (e.g., idle = 1800, current = 2100, distance = 300).
- **filtered_distance:** Distance after EWMA filtering.
- ***_8bits:** are the same values as the original variables but cast to 8 bits to limit the value range, reducing noise impact and enable a bit more efficient calculation.
- **velocity:** Rate of key press or release, calculated based on the difference in the new and old value of `filtered_distance_8bits`.

```
You, 5 days ago | 1 author (You)
struct __attribute__((__packed__)) state {
  uint16_t value;
  uint16_t distance;
  uint8_t distance_8bits;
  float filtered_distance;
  int8_t velocity;
  uint8_t filtered_distance_8bits;
};
```

Actuation Data

Stores the status and configuration of the key:

- **direction:** Current direction (up or down), inferred from the sign of velocity.
- **direction_changed_point:** Value recorded at the last direction change, used in rapid trigger logic.
- **status:** Key status, which can be:
  - `STATUS_MIGHT_BE_TAP`: Intermediate state, transitions to `STATUS_TAP` or `STATUS_TRIGGERED` depending on timing (if the key is pressed for less than a timeout value, it is a `STATUS_TAP`, otherwise it is a `STATUS_TRIGGERED`).
  - `STATUS_TAP`: One of two final status of a key, key press is sent and released immediately.
  - `STATUS_TRIGGERED`: One of two final status of a key, Key press is sent, but release may be delayed depending on if the user continue to hold the key or not.
  - `STATUS_RESET`: Default state when not pressed.
  - `STATUS_RAPID_TRIGGER_RESET`: A key will reach this status when it passes the rapid trigger reset threshold, which allows the key to enter `STATUS_TAP` or `STATUS_TRIGGERED` state again without returning to `STATUS_RESET` first.
- **trigger_offset:** Acts as the configurable actuation point. When distance exceeds this, the key is considered pressed.
- **reset_offset:** Defines the release point. When distance drops below this, the key is considered released.

  In mechanical switches, actuation and release points are usually the same. However, in this analog implementation, **reset_offset** is set slightly lower than **trigger_offset** to account for hysteresis, which prevents input chatter and accidental key toggles during small fluctuations around the threshold.
- **rapid_trigger_offset:** Similar to `trigger_offset`, but instead of being referenced from the `idle_value`, this offset is measured from the `direction_changed_point`, which is the position where the finger starts pressing down after moving upward (or vice versa) while still staying below the `triggered_offset`.

  This function enables faster re-triggering during rapid key taps or oscillations by dynamically adapting the actuation point based on movement direction, improving responsiveness.

```
You, 5 days ago | 1 author (You)
struct __attribute__((__packed__)) actuation {
  enum direction direction;
  uint8_t direction_changed_point;
  enum actuation_status status;
  uint8_t reset_offset;
  uint8_t trigger_offset;
  uint8_t rapid_trigger_offset;
  uint32_t triggered_at;
};
```

```
enum direction {
    GOING_UP,
    GOING_DOWN,
};
```

```
enum actuation_status {
    STATUS_MIGHT_BE_TAP,
    STATUS_TAP,
    STATUS_TRIGGERED,
    STATUS_RESET,
    STATUS_RAPID_TRIGGER_RESET
};
```

3.1.1. An example keyboard configuration

```c
struct key keyboard_keys[ADC_CHANNEL_COUNT][AMUX_CHANNEL_COUNT] = {0};
struct user_config keyboard_user_config = {
    .reverse_magnet_pole = 0,
    .trigger_offset = 64,
    .reset_threshold = 3,
    .rapid_trigger_offset = 40,
    .tap_timeout = 200,
    .keymaps = {
        // clang-format off
              [_BASE_LAYER] = {
                  {{HID_KEY_CONTROL_LEFT, HID_KEY_SHIFT_LEFT, HID_KEY_P, ____ },
                  {HID_KEY_4},
                  {HID_KEY_8, HID_KEY_9, HID_KEY_A, HID_KEY_B},
                  {HID_KEY_C, HID_KEY_D, HID_KEY_E, HID_KEY_F}},
                  {{HID_KEY_0, HID_KEY_1, HID_KEY_2, HID_KEY_3},
                  {HID_KEY_4, HID_KEY_5, HID_KEY_6, HID_KEY_7},
                  {HID_KEY_8, HID_KEY_9, HID_KEY_A, HID_KEY_B},
                  {HID_KEY_C, HID_KEY_D, HID_KEY_E, HID_KEY_F}},
                  {{HID_KEY_0, HID_KEY_1, HID_KEY_2, HID_KEY_3},
                  {HID_KEY_4, HID_KEY_5, HID_KEY_6, HID_KEY_7},
                  {HID_KEY_8, HID_KEY_9, HID_KEY_A, HID_KEY_B},
                  {HID_KEY_C, HID_KEY_D, HID_KEY_E, HID_KEY_F}},
                  {{HID_KEY_0, HID_KEY_1, HID_KEY_2, HID_KEY_3},
                  {HID_KEY_4, HID_KEY_5, HID_KEY_6, HID_KEY_7},
                  {HID_KEY_8, HID_KEY_9, HID_KEY_A, HID_KEY_B},
                  {HID_KEY_C, HID_KEY_D, HID_KEY_E, HID_KEY_F}},
              },
              [_TAP_LAYER] = {
                  {{____}, {____}, {____}, {____}},
                  {{____}, {____}, {____}, {____}},
                  {{____}, {____}, {____}, {____}},
                  {{____}, {____}, {____}, {____}},
              },
        // clang-format on
    }};
```

`reset_offset` is calculated on the basis of `trigger_offset` and `reset_threshold`:
`RESET_OFFSET` = `TRIGGER_OFFSET` − `RESET_THRESHOLD`

3.1.2. Functions of the module

Name	Description
`uint8_t get_bitmask_for_modifier (uint8_t keycode)`	Returns a bitmask with one bit set corresponding to the given modifier keycode (e.g., Ctrl, Shift). If the keycode is not a modifier, it returns 0x00. Used to identify and encode modifier keys.
`void init_key( uint8_t adc_channel, uint8_t amux_channel, uint8_t row, uint8_t column)`	Initializes a key in the `keyboard_keys` matrix at the given coordinates. Populates all fields and assigns keymaps to layers based on their type (NORMAL, MODIFIER, CONSUMER CONTROL, MACRO).
`uint8_t update_key_state( struct key *key)`	Reads the raw ADC value from the Hall effect sensor and performs an initial calibration during the first 20 cycles using an EWMA filter with = 0.6 to estimate the idle value. After calibration, the filter continues with = 0.8 to smooth out noise and reduce fluctuations. The function then calculates and updates the key's distance, filtered distance, velocity, and direction. It also tracks the maximum observed distance and the direction change point for use in rapid trigger logic.
`void update_key_actuation( struct key *key)`	Updates the actuation status of a key based on distance, direction, and thresholds and the current key state. Manages tap detection, key press/release calls, and transitions between RESET, TRIGGERED, TAP, and RAPID_TRIGGER_RESET.
`void update_key( struct key *key)`	Combines `update_key_state()` and `update_key_actuation()` to perform a complete update on a key's state and actuation logic. If the key is not ready (e.g. still calibrating), it does nothing.
`void keyboard_init_keys()`	Loops through the key matrix and initializes all keys using `init_key()` where where channel data is valid.(there is a channel mapped to that key).
`void keyboard_task()`	Main loop called in each cycle. Iterates through all enabled keys, reads input, updates state and actuation, and handles key press and hold logic.

3.2. USB

3.2.1. HID

Name	Description
`void hid_task()`	Sends HID reports (keyboard or consumer control) over USB when `should_send_*` flags are set and the USB device is ready.

`void hid_press_key(`
`struct key *key,`
`uint8_t layer)`
	Handles the logic to press a key. Depending on the key type (modifier, normal, macro, consumer control), it sets the corresponding USB HID report fields (`modifiers`, `keycodes[]`, or `consumer_report`) and sets flags to indicate a report should be sent. For macros, it processes both modifier and normal parts, ensuring they fit within the 6-key rollover limit.

`void hid_release_key(`
`struct key *key,`
`uint8_t layer)`
	Reverses the effect of `hid_press_key`, clearing the modifier bits, keycodes, or consumer report as appropriate. Also processes all keys in a macro, releasing both modifier and normal parts.

3.2.2. CDC

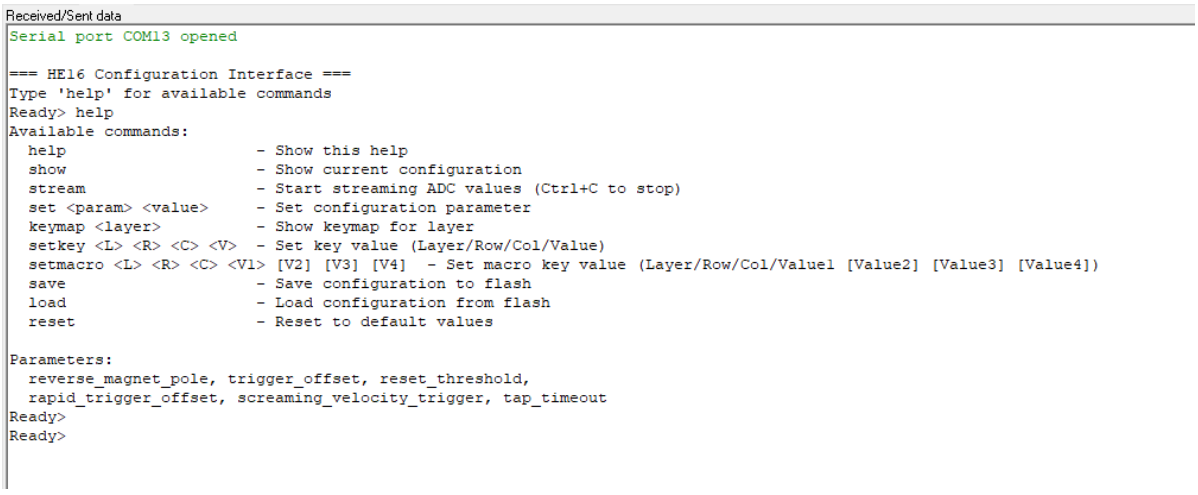


Figure 5: Available configuration and debugging commands

Name	Description
`void cdc_task(void)`	Main handler for USB CDC input. Reads and parses user input over serial. Handles control characters (backspace, enter, Ctrl+C), echoes input, and processes commands. Accommodates the basic functions of a terminal.
`void cdc_performance_measure( uint32_t started_at)`	Calculates the elapsed time since `started_at` using `HAL_GetTick()` and sends it to the host over CDC. Used to measure the performance of the system.
`static void handle_streaming(void)`	Sends periodic (1kHz) CSV-formatted ADC value of the sensors.
`static void start_streaming(void)`	Activates sensor value streaming and notifies the user.
`static void stop_streaming(void)`	Stops sensor value streaming.
`static void process_command(char *cmd)`	Parses a command string and dispatches it to corresponding actions like set, keymap, save, load, reset, etc.
`static void print_help(void)`	Sends a help message listing all available commands and configuration parameters.
`static void print_config(void)`	Sends the current `user_config` values (e.g. trigger thresholds, timeouts, etc.) to the user over CDC.
`static void set_config_value (char *param, char *value)`	Parses and sets a configuration parameter in the `keyboard_user_config` based on user input.
`static void print_keymap(uint8_t layer)`	Prints the full keymap of the given layer, including macro formatting for keys with multiple values.
`static void set_keymap_value( uint8_t layer, uint8_t row, uint8_t col, uint16_t value)`	Sets a single key's value in the keymap and clears any existing macro values. Also writes to flash and reinitializes keys.
`static void set_macro_keymap_value( uint8_t layer, uint8_t row, uint8_t col, uint16_t values[MAX_MACRO_LEN])`	Sets macro values for a key which contains multiple keycodes. Writes to flash and reinitializes the key matrix.
`static void save_config(void)`	Saves the current `user_config` to flash memory.
`static void load_config(void)`	Loads `user_config` from flash memory and reinitializes the key matrix.
`static void reset_config(void)`	Restores configuration to default values from `keyboard_default_user_config`, saves to flash, and reinitializes keys.
`static void cdc_write_string_chunked` (const char *str)	Writes a string to CDC output in manageable chunks to prevent buffer overflow.
`static void cdc_write_flush_wait(void)`	Flushes the CDC buffer and waits until all data is sent. Ensures message delivery.

Name	Description
`void ssd1306_Init(void)`	Initializes the OLED display by configuring addressing mode, orientation, contrast, charge pump settings, and display dimensions. Clears the buffer and flushes it to the screen.
`void ssd1306_UpdateScreen(void)`	Efficiently pushes only the changed parts of the screen buffer to the display via I^2C/SPI. Tracks modified pages and columns to reduce communication latency.
`void ssd1306_Fill(SSD1306_COLOR color)`	Fills the entire display buffer with the specified color (either all black or all white).
`void ssd1306_DrawPixel(uint8_t x, uint8_t y, SSD1306_COLOR color)`	Sets or clears a pixel at the given (x, y) coordinate in the buffer.
`char ssd1306_WriteChar(char ch, SSD1306_Font_t Font, SSD1306_COLOR color)`	Draws a single character using the specified font and color at the current cursor position in the screen buffer.
`char ssd1306_WriteString(char* str, SSD1306_Font_t Font, SSD1306_COLOR color)`	Writes a string to the buffer by printing characters one after another using the selected font.
`void ssd1306_SetCursor(uint8_t x, uint8_t y)`	Sets the internal cursor position (used by text functions) to the specified screen coordinates.
`void ssd1306_DrawRectangle(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color)`	Draws the outline of a rectangle using straight lines between corners.
`void ssd1306_FillRectangle(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color)`	Draws a filled rectangle of the given bounds and color.
`void ssd1306_DrawBitmap(uint8_t x, uint8_t y, const unsigned char* bitmap, uint8_t w, uint8_t h, SSD1306_COLOR color)`	Draws a bitmap image onto the screen buffer at the specified coordinates.

`void ssd1306_Line(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color)`	Draws a straight line between two points using Bresenham's line algorithm.
`void ssd1306_DrawCircle(uint8_t par_x, uint8_t par_y, uint8_t par_r, SSD1306_COLOR par_color)`	Draws the outline of a circle using integer math.
`void ssd1306_FillCircle(uint8_t par_x, uint8_t par_y, uint8_t par_r, SSD1306_COLOR par_color)`	Draws a filled circle by calculating pixel positions within the radius.
`void ssd1306_DrawArc(uint8_t x, uint8_t y, uint8_t radius, uint16_t start_angle, uint16_t sweep, SSD1306_COLOR color)`	Draws an arc (part of a circle) based on start and sweep angle.
`void ssd1306_DrawArcWithRadiusLine(uint8_t x, uint8_t y, uint8_t radius, uint16_t start_angle, uint16_t sweep, SSD1306_COLOR color)`	Draws an arc with connecting radius lines, forming a pie-slice shape.
`SSD1306_Error_t ssd1306_InvertRectangle(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2)`	Inverts the pixel colors in a rectangular region by toggling corresponding bits in the screen buffer.
`void ssd1306_SetContrast(uint8_t value)`	Adjusts the display contrast level by sending a command to the OLED controller.
`void ssd1306_SetDisplayOn(uint8_t on)`	Turns the display on or off without altering the buffer content.
`uint8_t ssd1306_GetDisplayOn(void)`	Returns the current display on/off state.
`void ssd1306_FlipScreen(uint8_t horz, uint8_t vert)`	Changes the horizontal and/or vertical scan direction to flip the screen content.

Each update cycle displays up to six actively pressed keys—three on the top row and three on the bottom. For each key, its hexadecimal keycode and actuation depth (in percentage) are shown. The percentage value is computed from the final filtered analog reading, which results from combining an RC low-pass hardware filter with an

EWMA (Exponentially Weighted Moving Average) software filter.

```
for (int i = 1; i <= 3; i++) {
  if (keycodes[i - 1][0] != '\0') {
    int x = MOD_WIDTH + (i - 1) * KEY_WIDTH + 4;
    ssd1306_SetCursor(x, label_row_top);
    ssd1306_WriteString((char *)keycodes[i - 1], Font_6x8, Black);

    char buf[6];
    sprintf(buf, "%d%%", key_percents[i - 1]);
    ssd1306_SetCursor(x, percent_row_top);
    ssd1306_WriteString(buf, Font_6x8, Black);
  }
}

for (int i = 4; i <= 6; i++) {
  if (keycodes[i - 1][0] != '\0') {
    int x = MOD_WIDTH + (i - 4) * KEY_WIDTH + 4;
    ssd1306_SetCursor(x, label_row_bot);
    ssd1306_WriteString((char *)keycodes[i - 1], Font_6x8, Black);

    char buf[6];
    sprintf(buf, "%d%%", key_percents[i - 1]);
    ssd1306_SetCursor(x, percent_row_bot);
    ssd1306_WriteString(buf, Font_6x8, Black);
  }
}
```

Modifier keys such as LCtrl, RShift, and LAlt are detected via bitmask values and displayed on the left side of the screen. Each active modifier is printed on a new line with vertical spacing managed programmatically.

```
if (label) {
    ssd1306_SetCursor(2, mod_y);
    ssd1306_WriteString(label, Font_6x8, Black);
    mod_y += mod_line_height;
}
```

Each cycle scans all analog key channels to determine which keys are actively pressed and whether they are standard keys or modifiers. For each active key (with distance_8bits 15), the system first checks whether the key type is a modifier. If so, it reads the bitmask from the key's configuration and maps it to a human-readable label (e.g., LCtrl, RShift). If the key is a normal key and fewer than six standard keys have been detected so far, the system stores a hexadecimal representation of the key's AMUX index. It also calculates the actuation percentage based on the distance_8bits value for later presentation.

```
for (int amux = 0; amux < AMUX_CHANNEL_COUNT; amux++) {
        struct key* k = &keyboard_keys[0][amux];

        if (k->state.distance_8bits >= 15 && k->layers[_BASE_LAYER].type == KEY_TYPE_MODIFIER) {
            uint16_t bitmask = *(uint16_t *)k->layers[_BASE_LAYER].value;
            const char* label = NULL;

            if (bitmask == 0b00000001) label = "LCtrl";
            else if (bitmask == 0b00000010) label = "LShift";
            else if (bitmask == 0b00000100) label = "LAlt";
            else if (bitmask == 0b00001000) label = "LGUI";
            else if (bitmask == 0b00010000) label = "RCtrl";
            else if (bitmask == 0b00100000) label = "RShift";
            else if (bitmask == 0b01000000) label = "RAlt";
            else if (bitmask == 0b10000000) label = "RGUI";

            if (label) {
                ssd1306_SetCursor(2, mod_y);
                ssd1306_WriteString(label, Font_6x8, Black);
                mod_y += mod_line_height;
            }
        }

        else if (k->state.distance_8bits >= 15 && tracker < 6 && k->layers[_BASE_LAYER].type == KEY_TYPE_NOR
            keycodes[tracker][0] = '0';
            keycodes[tracker][1] = 'x';
            keycodes[tracker][2] = (amux < 10) ? ('0' + amux) : ('A' + (amux - 10));
            keycodes[tracker][3] = '\0';

            key_percents[tracker] = (k->state.distance_8bits * 100) / 255;
            tracker++;
        }
    }
```

3.4. Result

The processing time for one cycle (time taken to through all tasks once) can be calculated by getting the time difference between the start and end of the main while loop. After that, the result is printed to the terminal.

```
while (1) {
  // MARK: Main loop
  start_at=HAL_GetTick();
  tud_task();
  keyboard_task();
  hid_task();
  cdc_task();
  cdc_performance_measure(start_at);
```

Figure 6: while loop code snippet

```
One cycle duration: 1
One cycle duration: 0
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 0
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 0
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 0
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
One cycle duration: 0
One cycle duration: 1
One cycle duration: 1
One cycle duration: 1
```

Figure 7: Cycle duration measurement

The average duration of one cycle is much less than 1 tick (1ms), which satisfies the condition of 1kHz keyboard polling rate.

By using the debugging interface exposed through the Virtual COM Port, the final filtered sensor reading (RC low pass hardware filter + EWMA software filter) can be viewed.

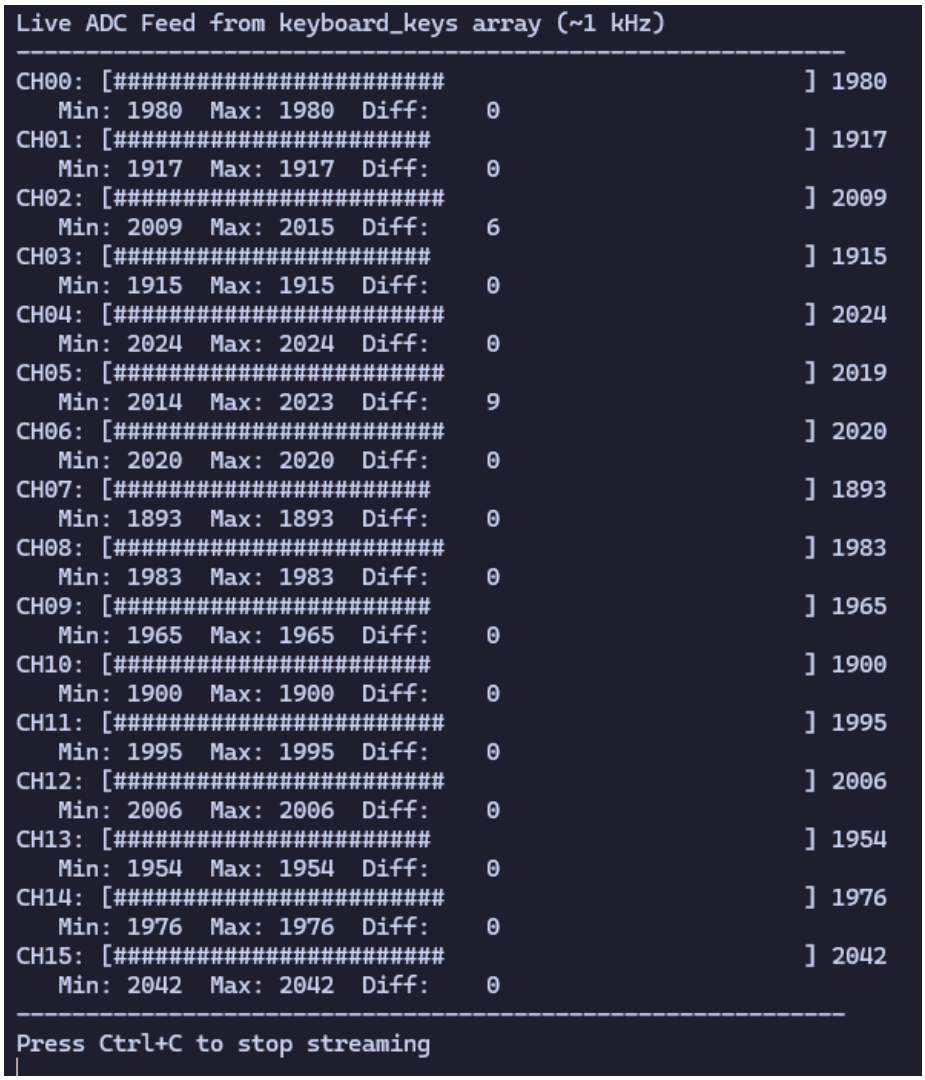


Figure 8: Value of sensors at idle state

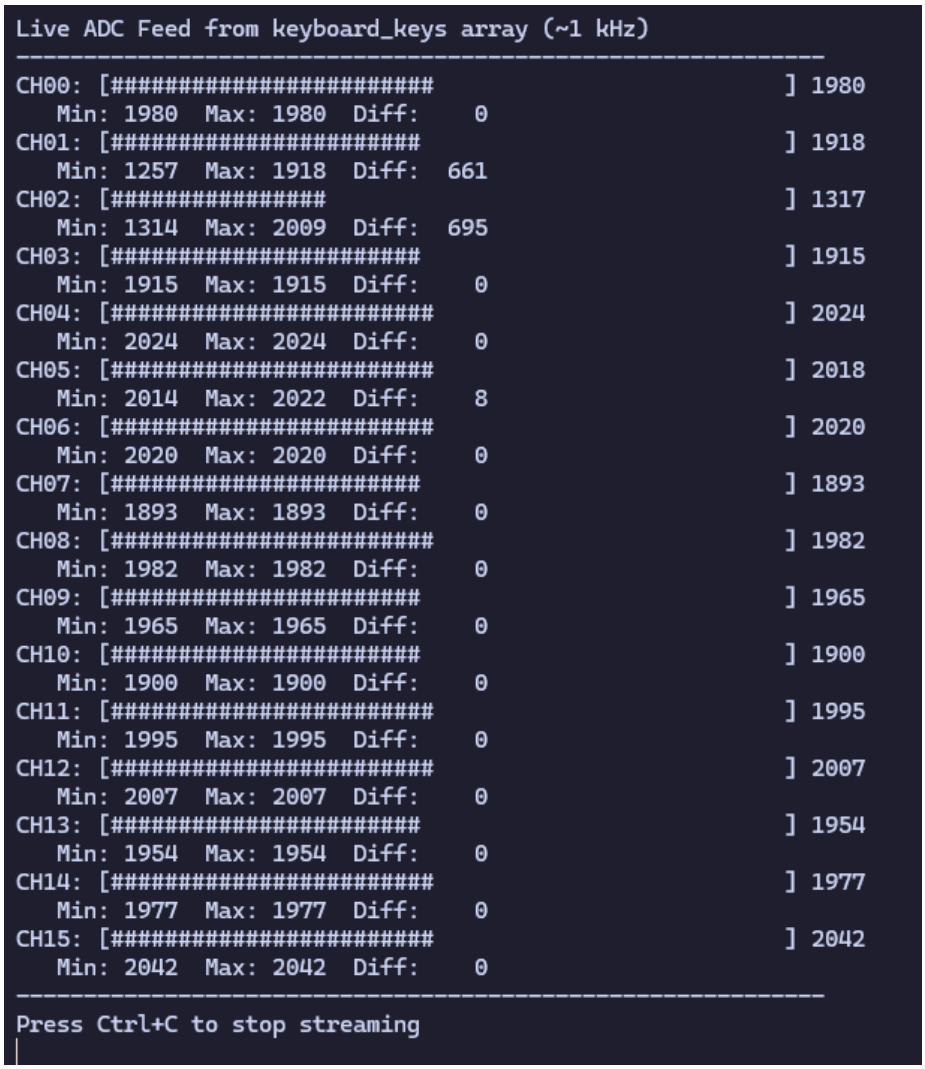


Figure 9: Value of sensors when fully pressed

The voltage fluctuation at `idle_state` is very low after filtering.
Both polling rate and sensor output measurement are within the expected value defined in Non-Functional Requirements section.

The complete system test is uploaded to Teams alongside the report.

3.5. Contribution

Name	**Task**
Le Ngoc Quang Hung	Hardware and software designs. Codebase initialization. Handle code for setup and descriptors of the USB module Handle code for reading analog input and update key state Handle code for trigger and macro functions for keyboard module
Tuong Phi Tuan	Handle code for OLED Display Optimized the SSD1306 screen update routine
Tran Quang Hung	Handle code for snaptap functions Handle code for macro functions

3.6. Evaluation and Assessment

The system has met all defined functional and non-functional requirements and functions reliably. All core features, including analog key sensing using Hall effect sensors, dynamic actuation point handling (rapid trigger), macro function, auto calibration method and USB Composite Device communication are working as intended. The firmware handles real-time key updates efficiently, and the modular design allows for future expandability, such as additional layers and keys in the software or adding new hardwares such as rotary encoders.

However, the biggest hurdle during development was the late arrival of the PCB, which significantly delayed hardware testing and integration. This bottleneck forced much of the software development and debugging process to be done on a breadboard, resulting in limited time for full system validation once the hardware was finally available.

Another limitation is the current configuration interface, which only accepts keycodes in decimal form. While functional, this reduces usability—especially for users more familiar with symbolic key representations (e.g., `KC_A, CTRL, or MEDIA_PLAY`).

Overall, the system demonstrates a reliable and well-structured implementation of a custom keyboard firmware. It successfully integrates analog key sensing with Hall-Effect sensors, dynamic actuation logic, macro capabilities, and USB Composite Device communication. The architecture is efficient and modular, supporting smooth real-time key updates and leaving ample room for expansion.