kzooting: Un teclado analógico programable

Isaac A. Castro y Carlos A. Castro

Facultad de Electrotecnia y Computación, Universidad Nacional de Ingeniería

Máquinas y Computadoras

Ing. Jairo Fúnez Lira

7 de Julio del 2023

kzooting: Un teclado analógico reprogramable

El kzooting es un teclado programable de entrada analógica que cuenta con una implementación de la tecnología de *rapid trigger* desarrollada por la empresa wooting y una implementación de macros. Para el teclado se utilizó como microcontrolador una Raspberry Pi Pico.

La tecnología de rapid trigger permite al teclado detectar con mayor sensibilidad y recurrencia al presionamiento y liberación de una tecla; Para esto se necesita de un teclado que reciba el estado de la tecla de forma analógica en vez de digital. En esencia, hace uso de la posición de la tecla para detectar los cambios en su movimiento y usar estos cambios para detectar la actuación de la tecla en vez de ocupar un punto de actuación predefinido.

“Elimina el segundo elemento más lento en latencia de input; el movimiento necesario para liberar una tecla. El Rapid Trigger cambia dinámicamente el punto de actuación y liberación. Tus teclas se activarán cuando desees presionarlas y se liberarán cuando desees dejarlas ir.” (Wooting, n.d.)

“10/10. El Wooting 60HE es el teclado para juegos más rápido que existe y logra superar la salida de la mayoría de marcas establecidas.” (Dexerto, 2022)

Los teclados de wooting envían a la computadora los valores analógicos de sus teclas por lo que necesitan de un driver especializado para poder trabajar y utilizar el rapid trigger. En cambio, en nuestro kzooting se optó por ejecutar toda la lógica requerida para esto desde el microcontrolador del teclado, permitiendo así que se pueda usar un driver genérico para este.

En el teclado kzooting, los switches utilizan sensores de efecto Hall, un efecto que permite medir la intensidad de un campo magnético, para detectar la altura de la tecla con respecto a la base.

El aspecto reprogramable del kzooting se realiza abriendo un puerto serial de comunicación con la computadora. En este, hemos definido diversos comandos que la computadora puede enviar al kzooting. Ya sea para conocer la configuración actual, la temperatura, etc. O para actualizar el mapa actual de las teclas. El intercambio de información se hace usando una forma de serialización de datos llamada JSON. El kzooting admite que sus teclas puedan ser programadas tanto a teclas únicas como combinaciones de teclas o a secuencias de las mismas, es decir macros.

Objetivos

General:

- Crear un teclado programable con entrada analógica.

Específicos:

- Analizar la tecnología *rapid trigger* de los teclados wooting.
- Definir una estructura de código que permita implementar macros.
- Diseñar una aplicación de escritorio para configurar el teclado con facilidad.

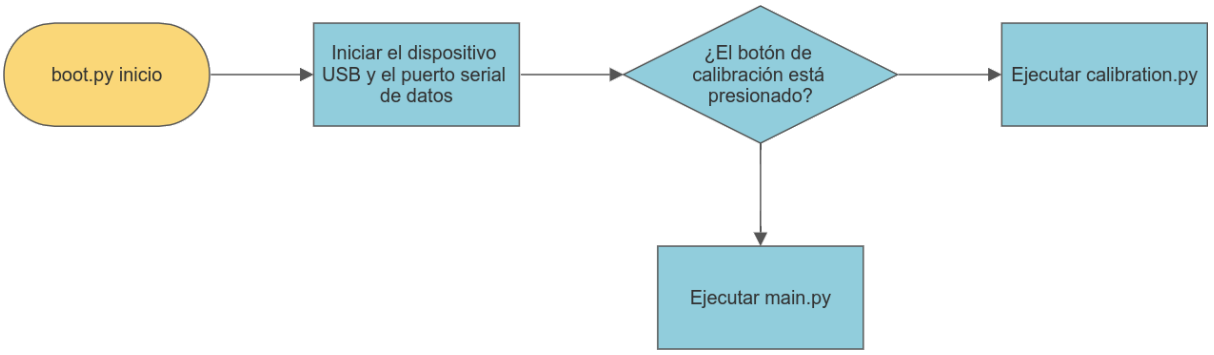
Alcances

- Modificar switches digitales CHERRY MX Red para volverlos analógicos, esto se logrará agregando un imán de neodimio en la parte de abajo del stem y usando un sensor de efecto hall por debajo de todo el switch que detecte qué tan cerca está el imán antes mencionado, de esta forma, mientras más se presione el switch, tendremos una señal más fuerte.
- Implementar la tecnología rapid trigger inspirada en los teclados de la compañía wooting con la cual se puede tener un punto de activación dinámico para cada switch, haciendo que el teclado sea más responsivo que cualquier teclado digital. Esta tecnología está orientada a las personas que quieren un teclado para uso en juegos competitivos.
- Elaborar un teclado USB 1.1 de 9 teclas, conectando la entrada Micro USB de una RP2 a un ordenador mediante USB A.
- El teclado será programable, y tendrá las siguientes funcionalidades
    - Macros: Cada tecla tendrá la opción de que su acción sea enviar cadenas de teclas
    - Sensibilidad: Se podrá cambiar la sensibilidad para el rapid trigger con un mínimo de 0.15mm.
    - Punto de actuación: Se dará la opción de no usar rapid trigger, en cuyo caso el usuario podrá cambiar el punto fijo de activación del switch a la distancia que desee.

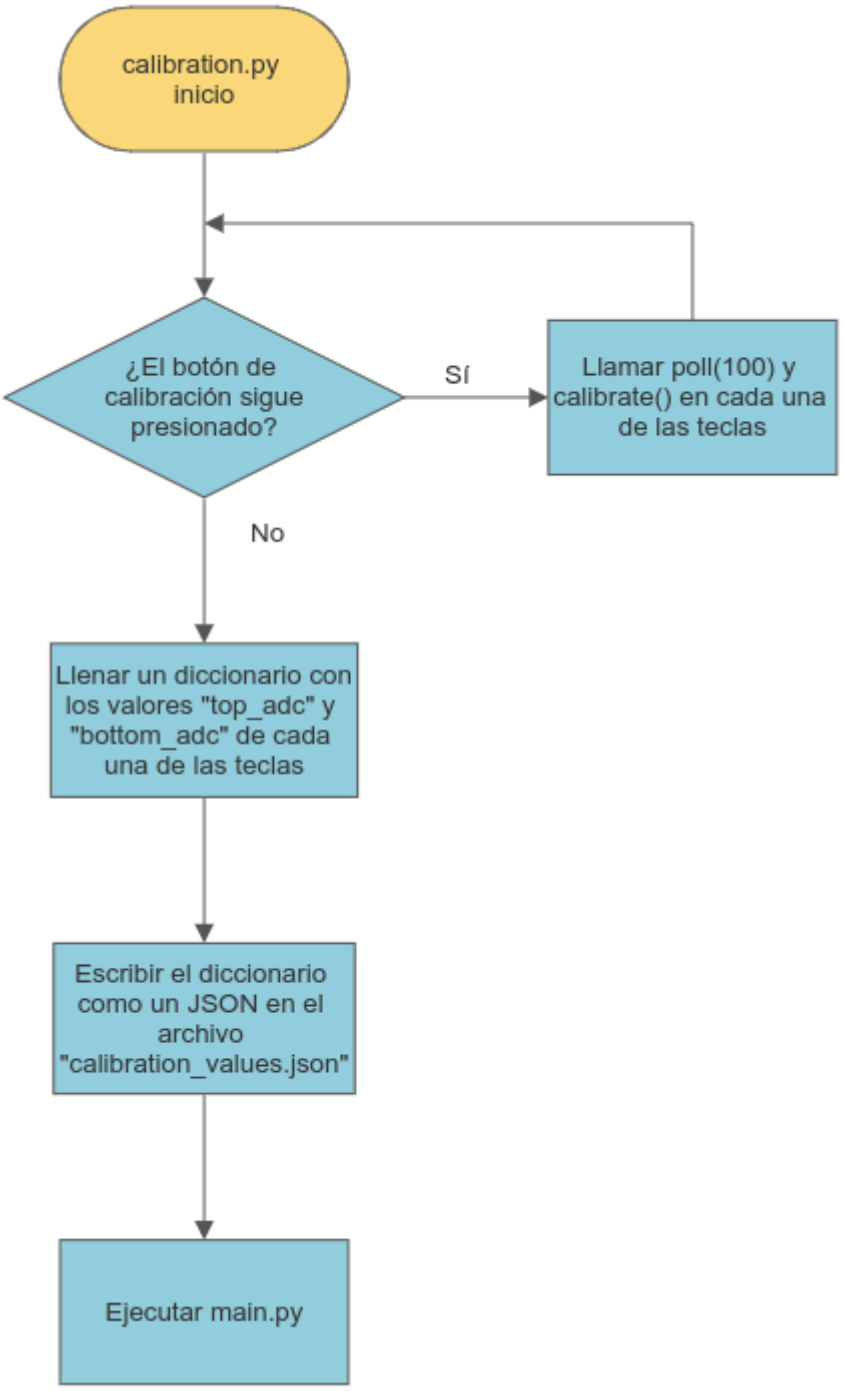
Diagramas de flujo

A continuación se presentan los diagramas de flujo del código empleado en la Raspberry Pi Pico:

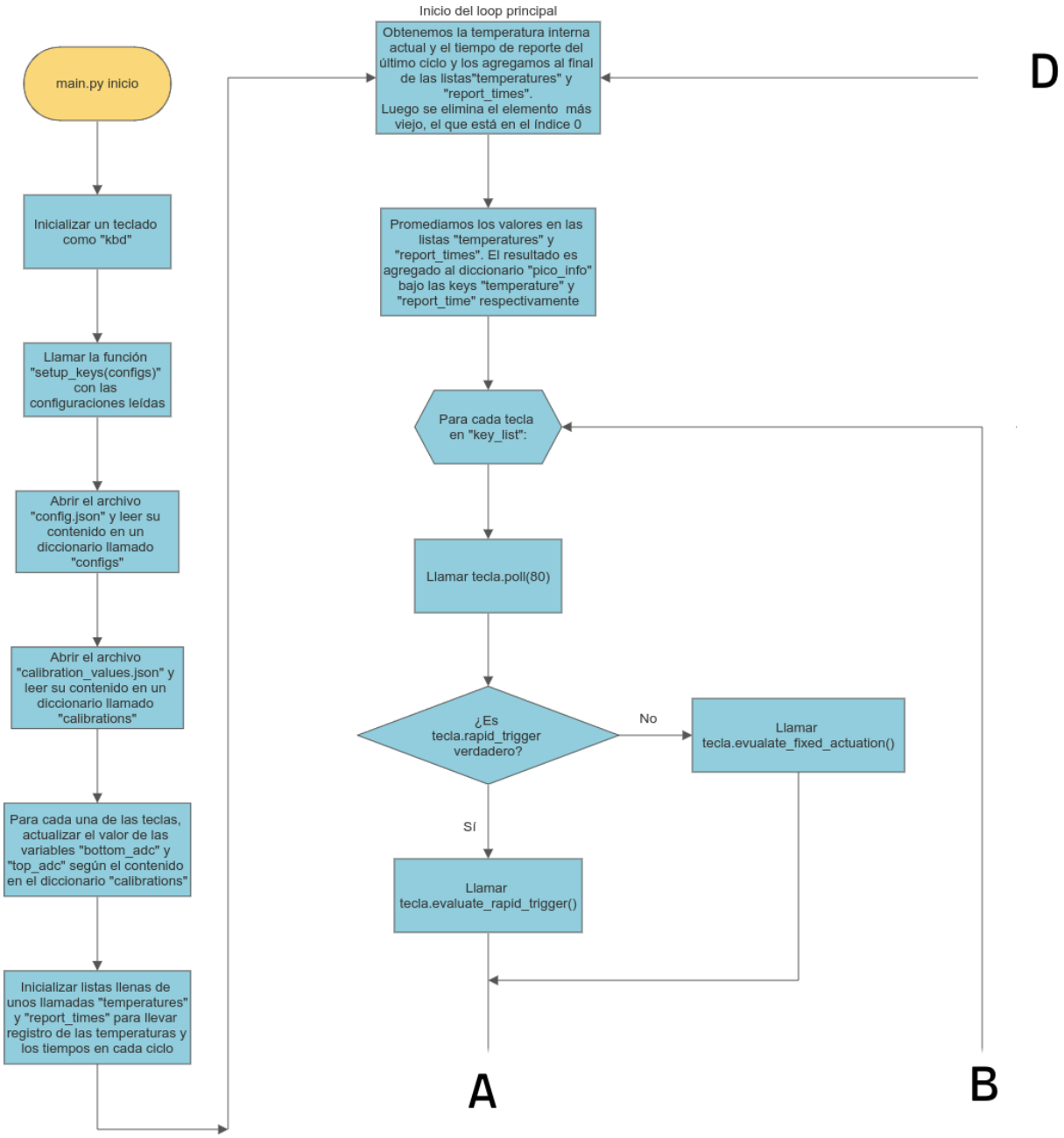
boot.py



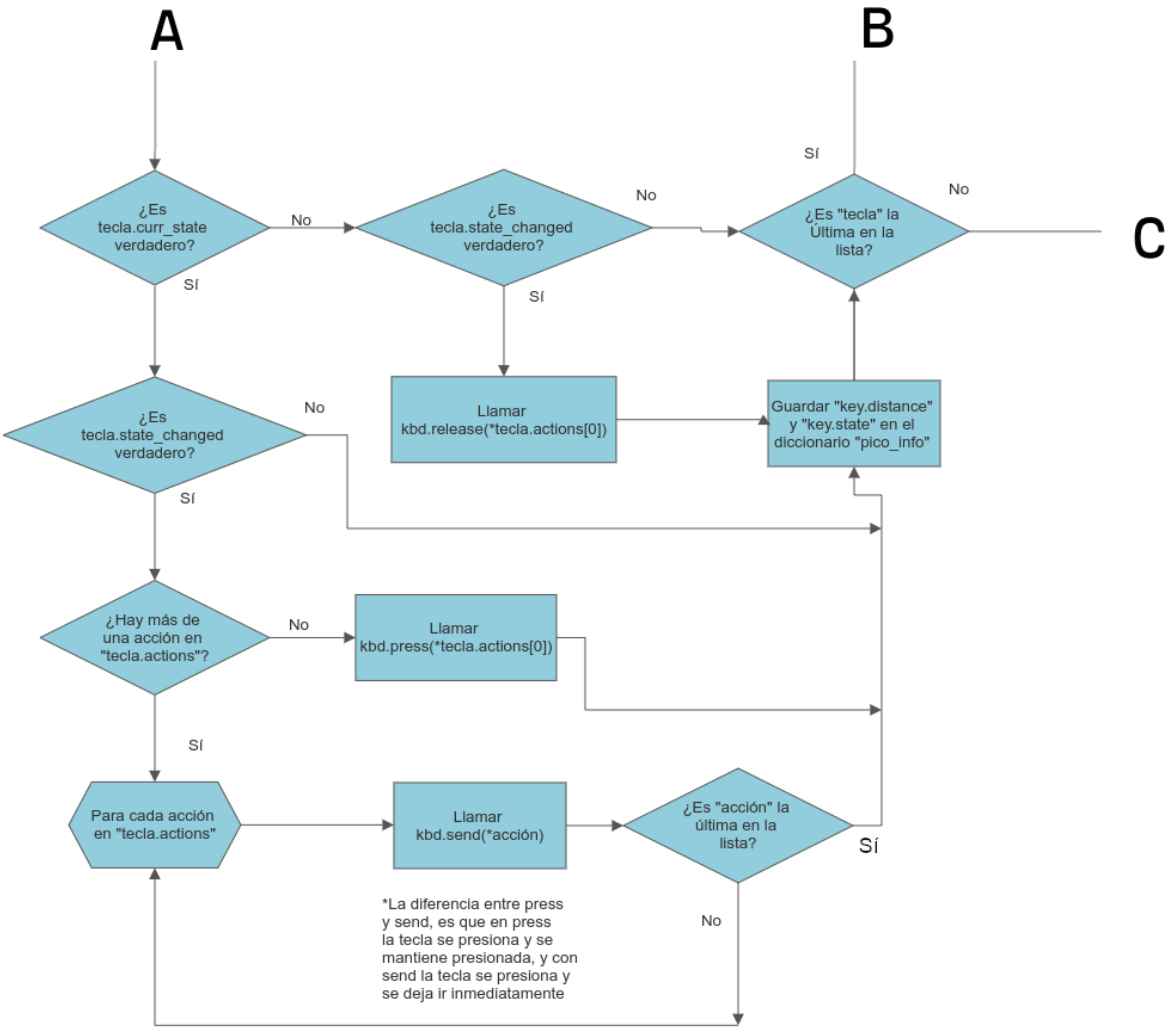
calibration.py



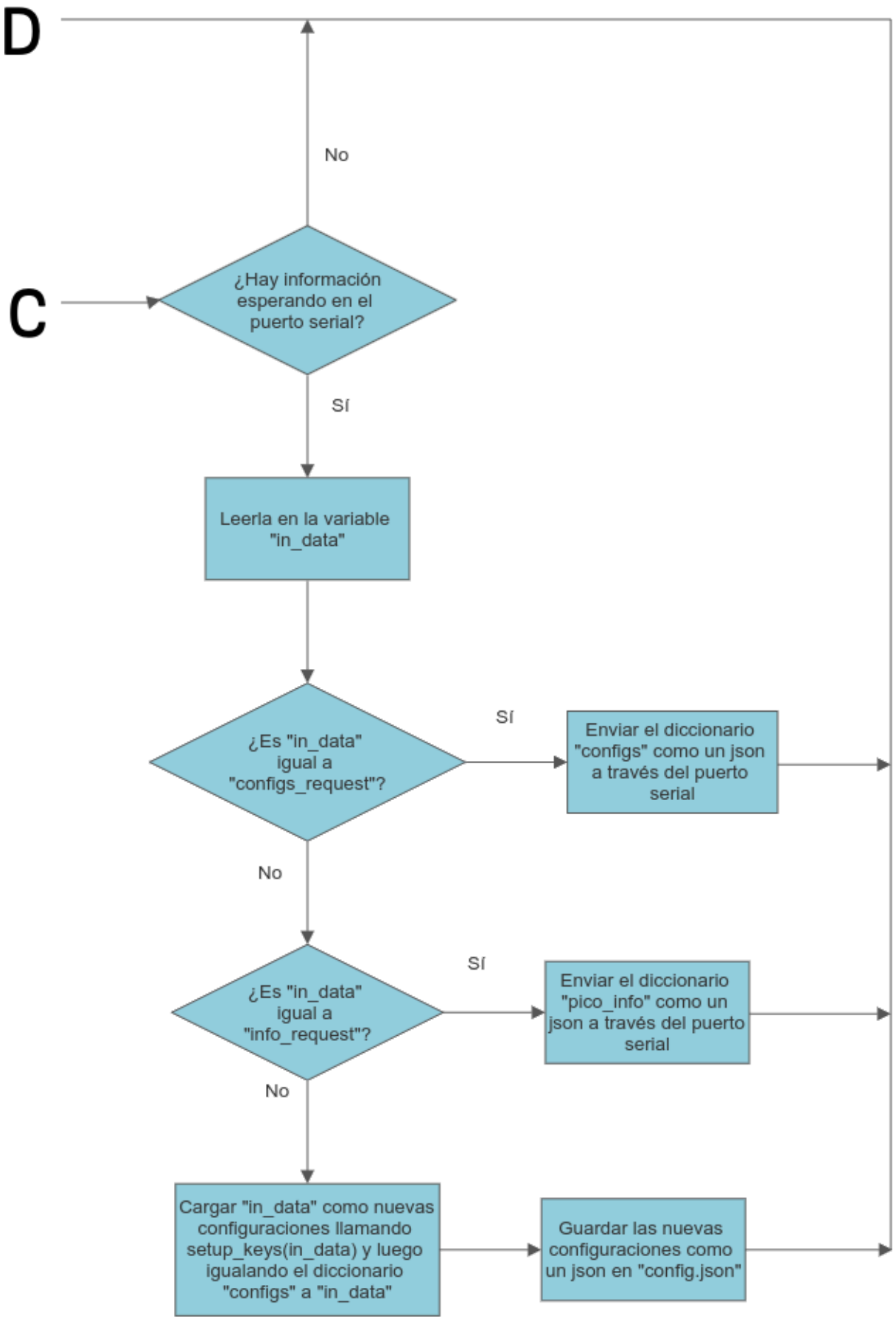
main.py (función main)



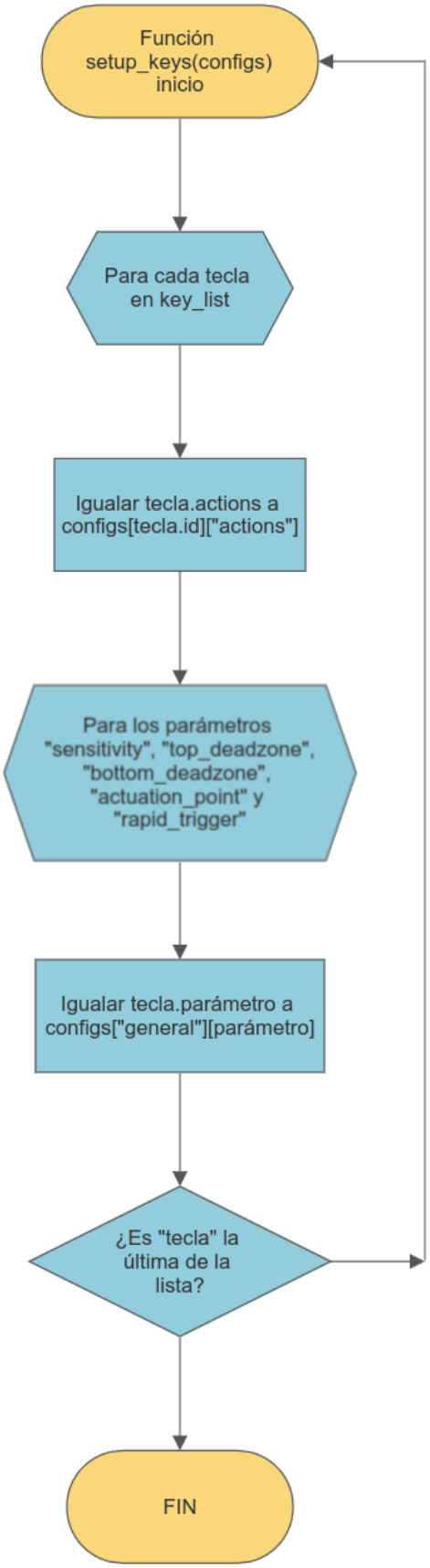
Continuación hacia abajo



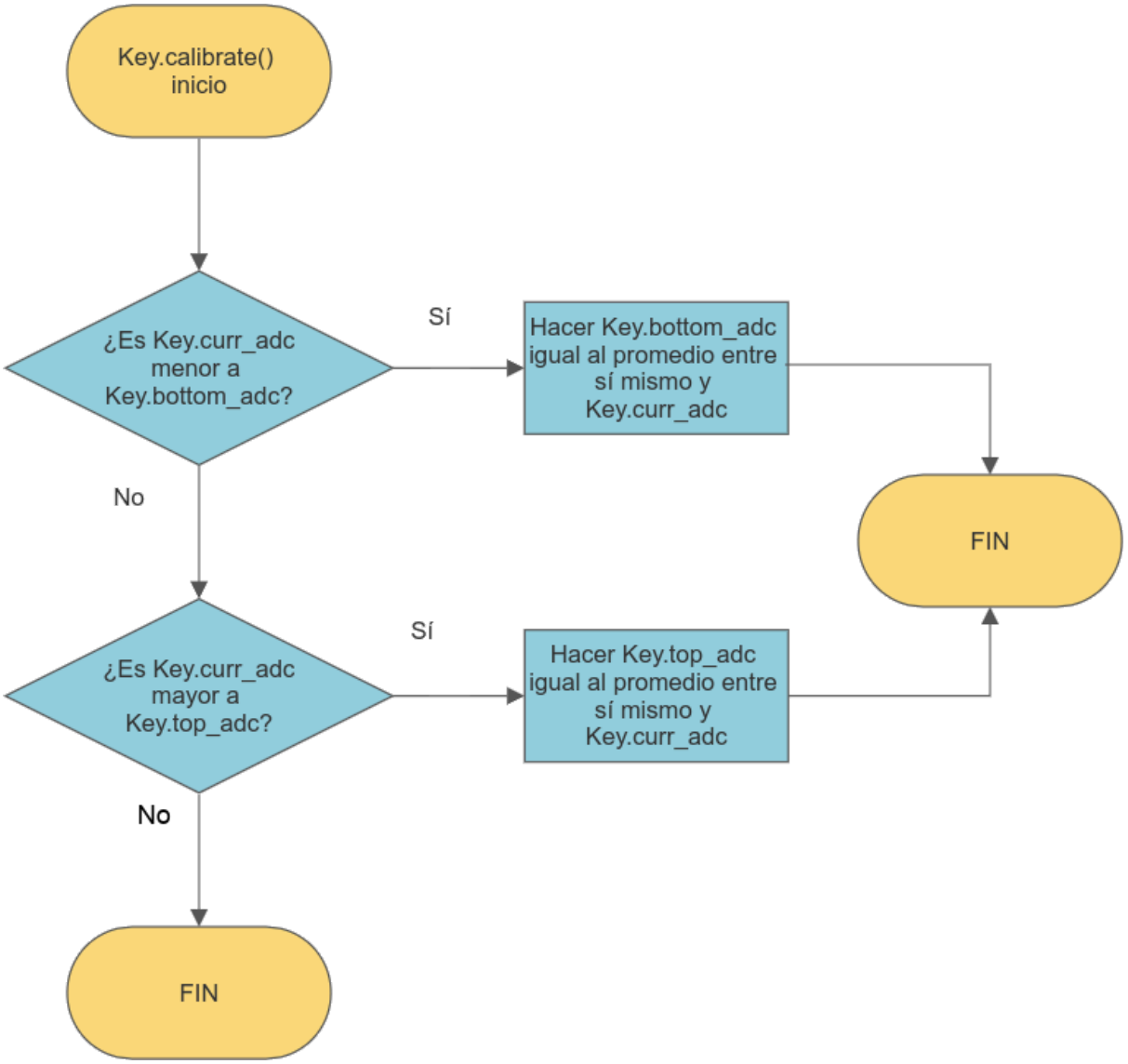
Continuación hacia la derecha

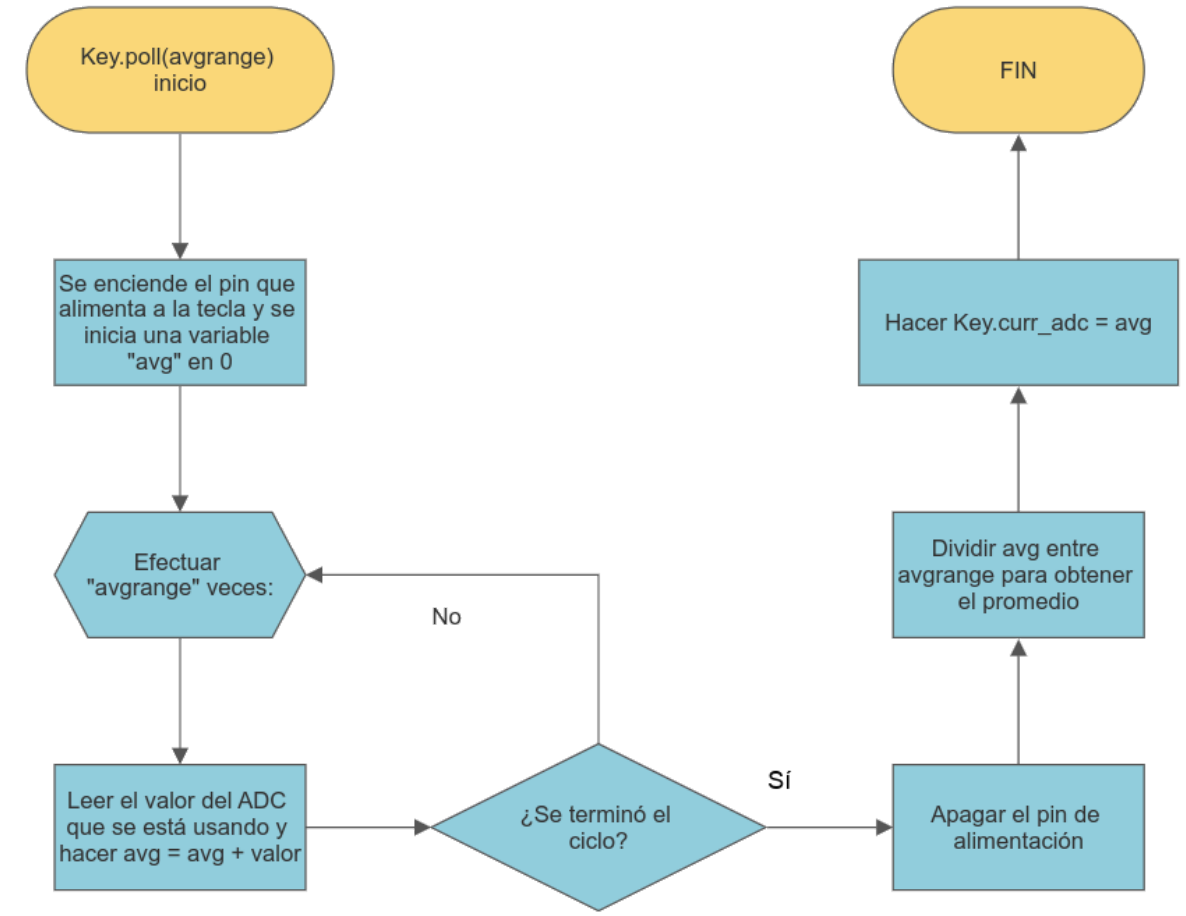


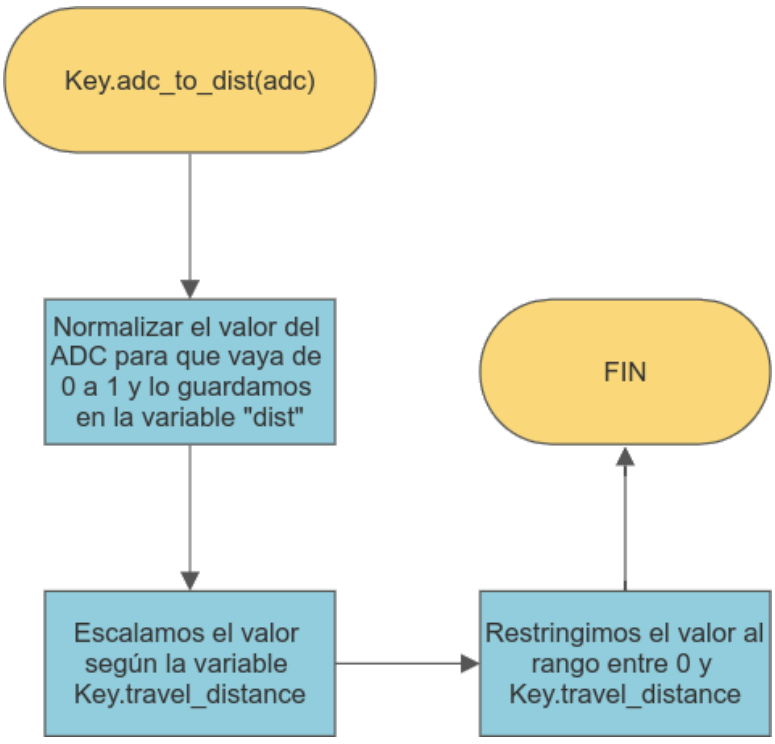
main.py (función setup_keys)

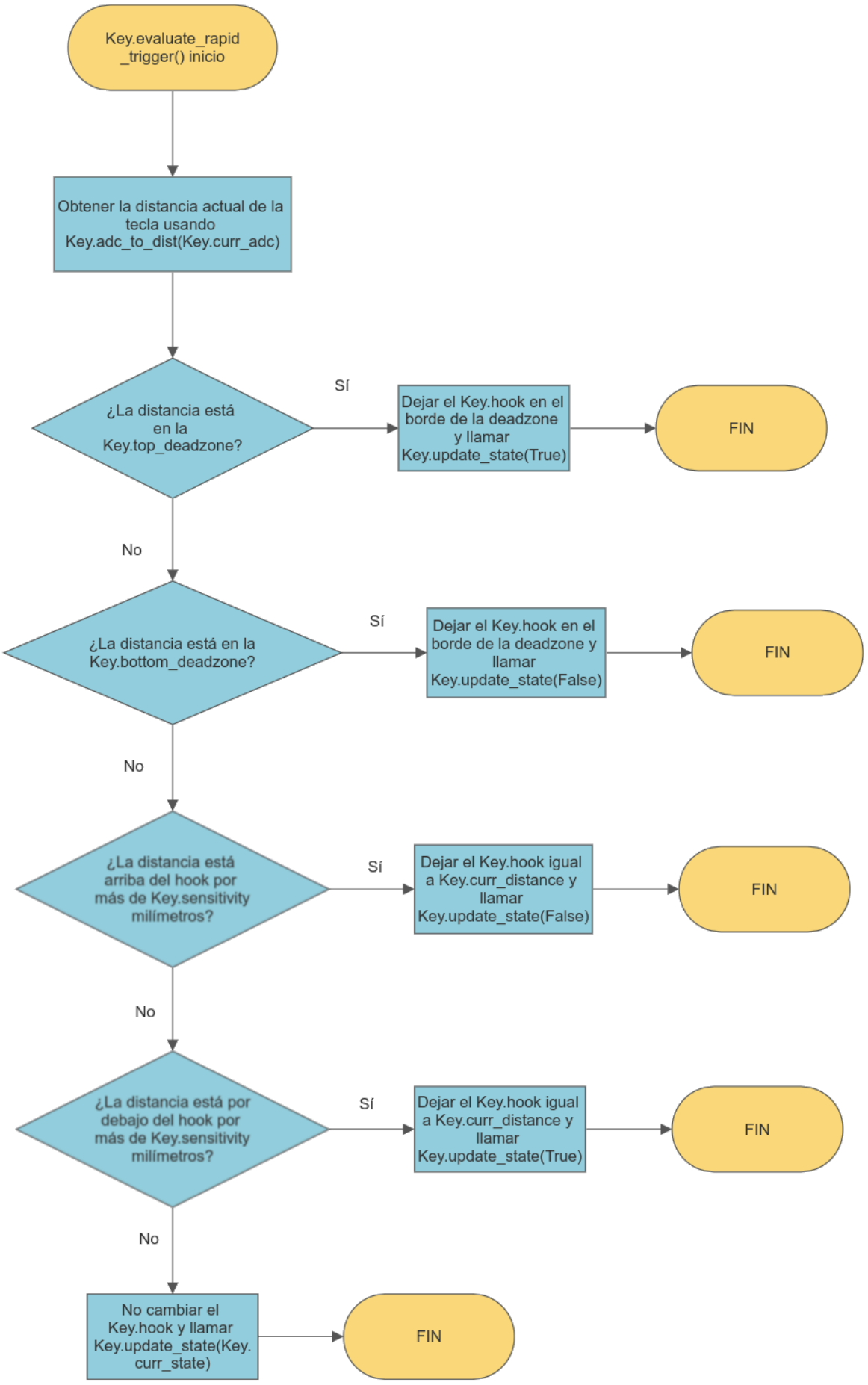


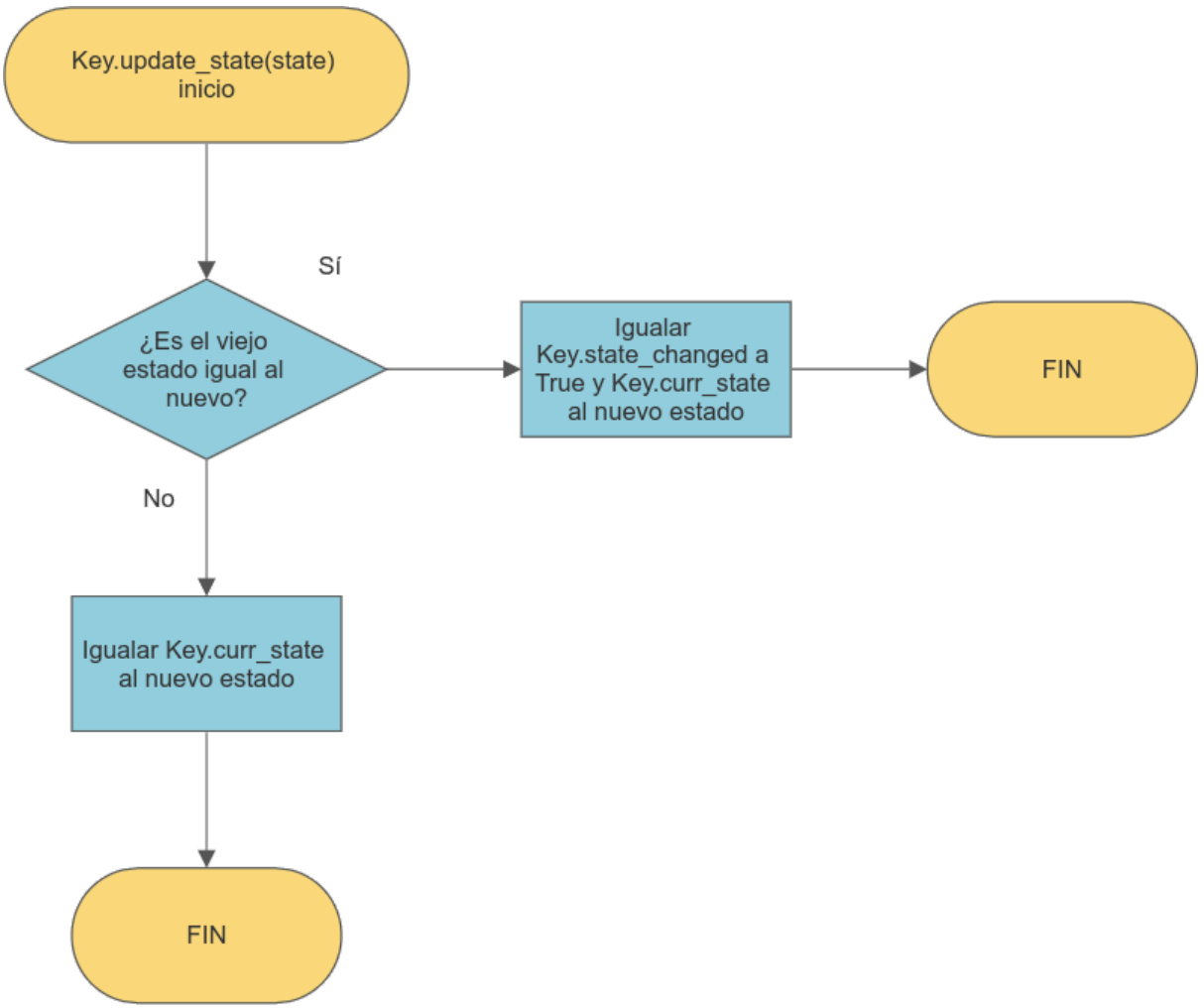
Métodos de la clase Key




Key.poll(avgrange)
inicio
Se enciende el pin que alimenta a la tecla y se inicia una variable "avg" en 0
Efectuar "avgrange" veces:
Leer el valor del ADC que se está usando y hacer avg = avg + valor
¿Se terminó el ciclo?
No
Sí
Apagar el pin de alimentación
Dividir avg entre avgrange para obtener el promedio
Hacer Key.curr_adc = avg
FIN


Key.adc_to_dist(adc)
Normalizar el valor del ADC para que vaya de 0 a 1 y lo guardamos en la variable "dist"
Escalamos el valor según la variable Key.travel_distance
Restringimos el valor al rango entre 0 y Key.travel_distance
FIN


Key.evaluate_rapid
_trigger() inicio
Obtener la distancia actual de la tecla usando Key.adc_to_dist(Key.curr_adc)
¿La distancia está en la Key.top_deadzone?
Sí
Dejar el Key.hook en el borde de la deadzone y llamar Key.update_state(True)
FIN
No
¿La distancia está en la Key.bottom_deadzone?
Sí
Dejar el Key.hook en el borde de la deadzone y llamar Key.update_state(False)
FIN
No
¿La distancia está arriba del hook por más de Key.sensitivity milímetros?
Sí
Dejar el Key.hook igual a Key.curr_distance y llamar Key.update_state(False)
FIN
No
¿La distancia está por debajo del hook por más de Key.sensitivity milímetros?
Sí
Dejar el Key.hook igual a Key.curr_distance y llamar Key.update_state(True)
FIN
No
No cambiar el Key.hook y llamar Key.update_state(Key.curr_state)
FIN


Key.update_state(state)
inicio
Sí
¿Es el viejo estado igual al nuevo?
Igualar Key.state_changed a True y Key.curr_state al nuevo estado
FIN
No
Igualar Key.curr_state al nuevo estado
FIN

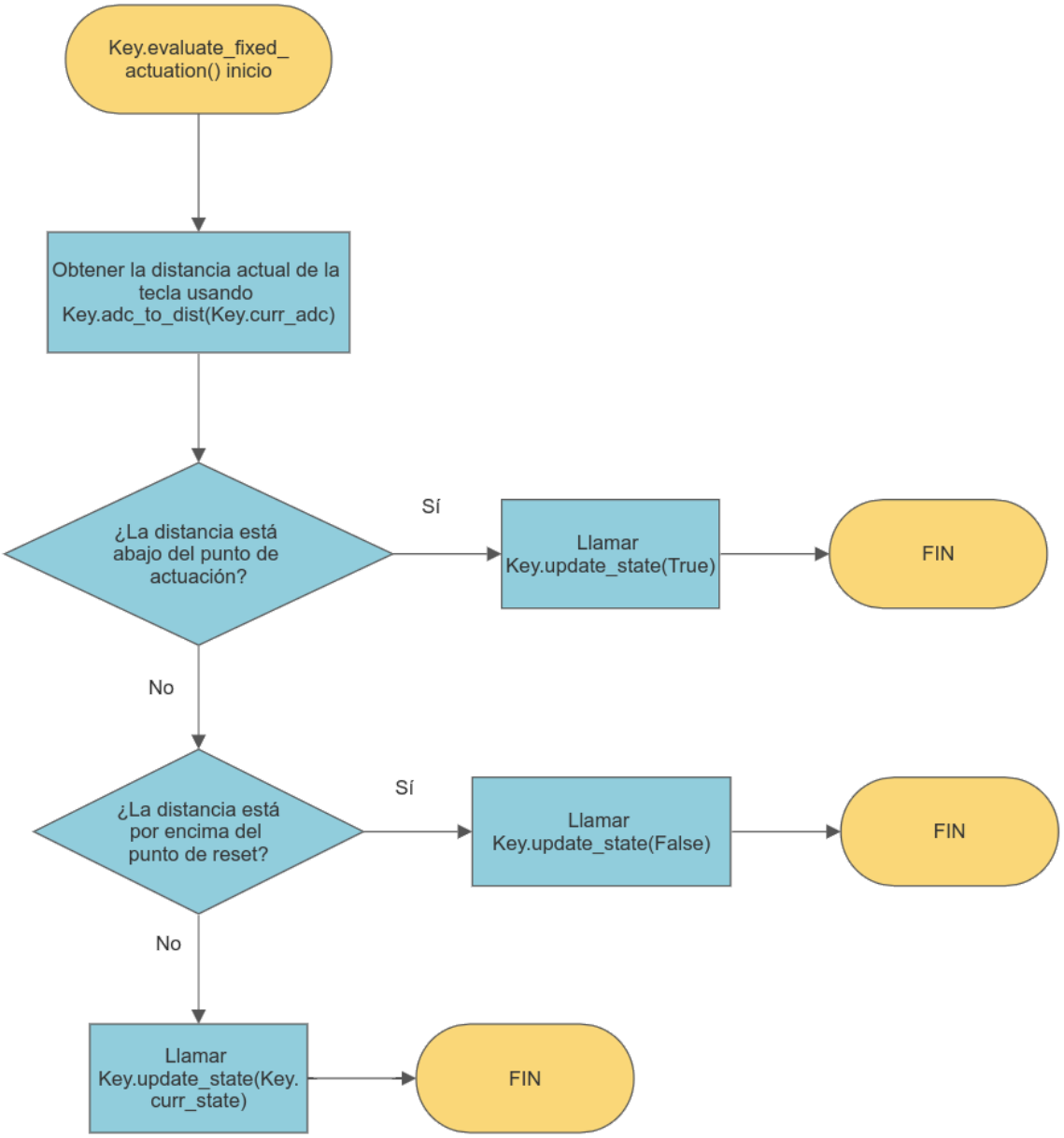
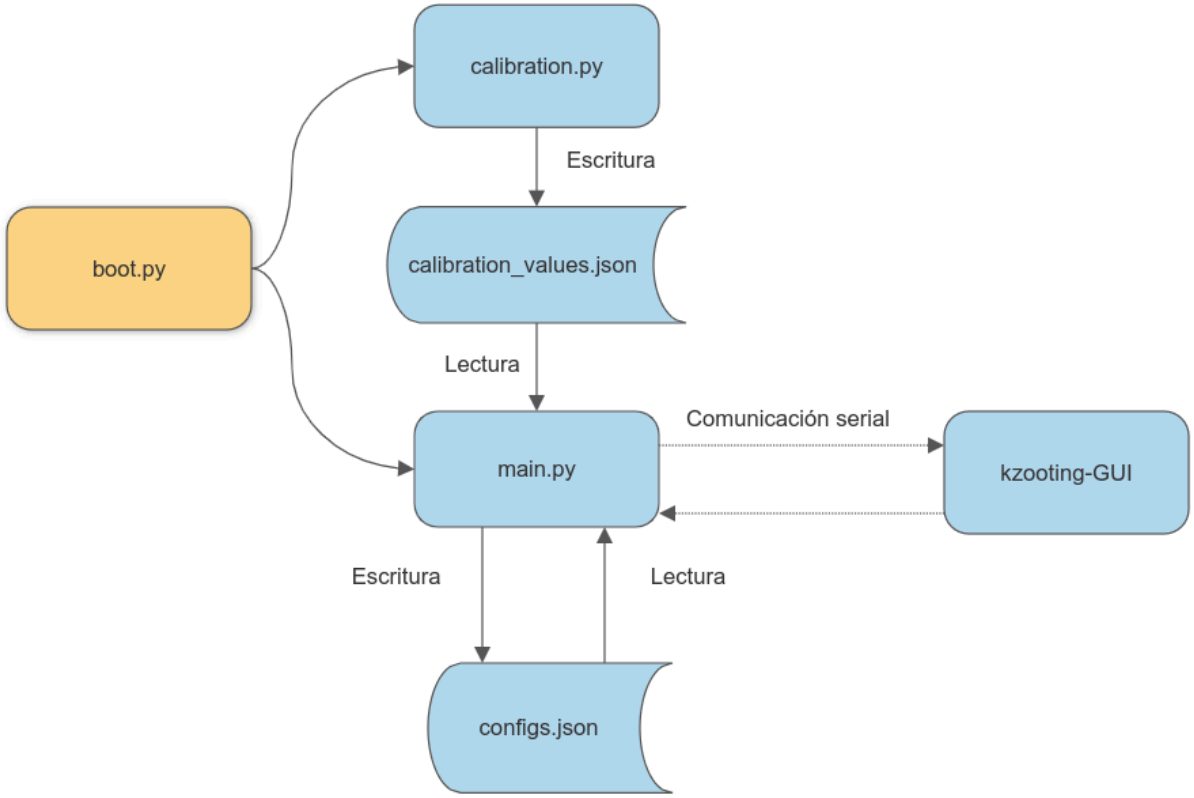

Key.evaluate_fixed_
actuation() inicio
Obtener la distancia actual de la tecla usando Key.adc_to_dist(Key.curr_adc)
¿La distancia está abajo del punto de actuación?
Sí
Llamar Key.update_state(True)
FIN
No
¿La distancia está por encima del punto de reset?
Sí
Llamar Key.update_state(False)
FIN
No
Llamar Key.update_state(Key.curr_state)
FIN

Diagrama de bloques



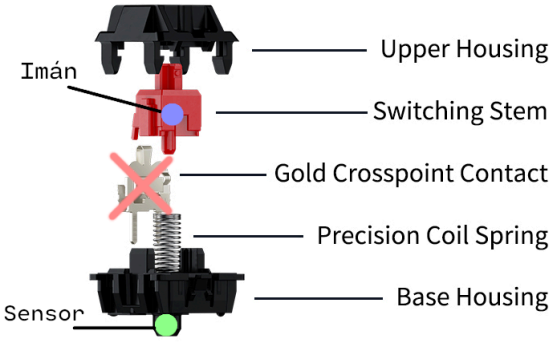
´´

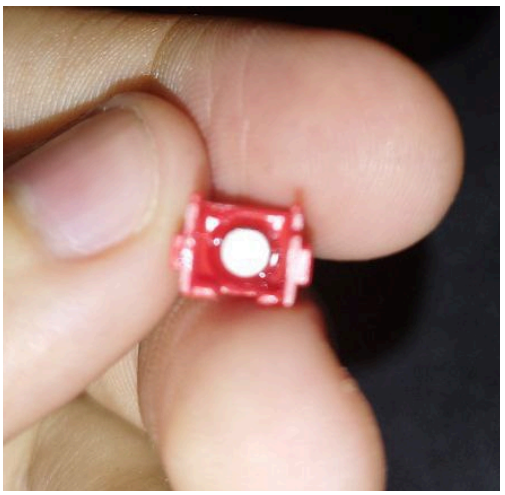
Elaboración del proyecto

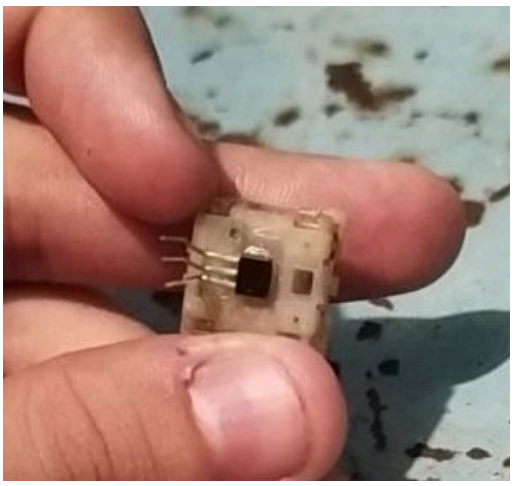
Se comenzó por conseguir los materiales principales, los cuales fueron:

- Imanes de neodimio de 3x2mm
- Sensores de efecto hall lineales OH49E
- Switches mecánicos CHERRY MX Red

Se procedió a modificar los switches digitales para volverlos analógicos. Este proceso consiste en cortar una parte del *stem* para agregar un imán de neodimio, y cortar una parte del *base housing* para agregar el sensor de efecto hall. También se removieron los contactos de cobre debido a que eran innecesarios.

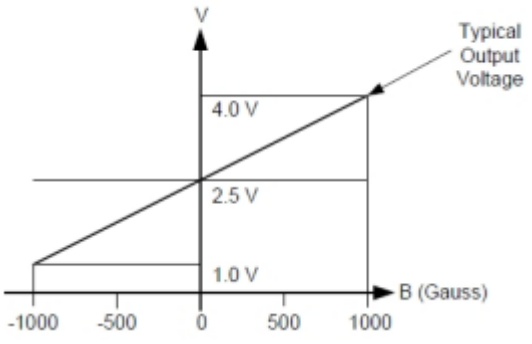






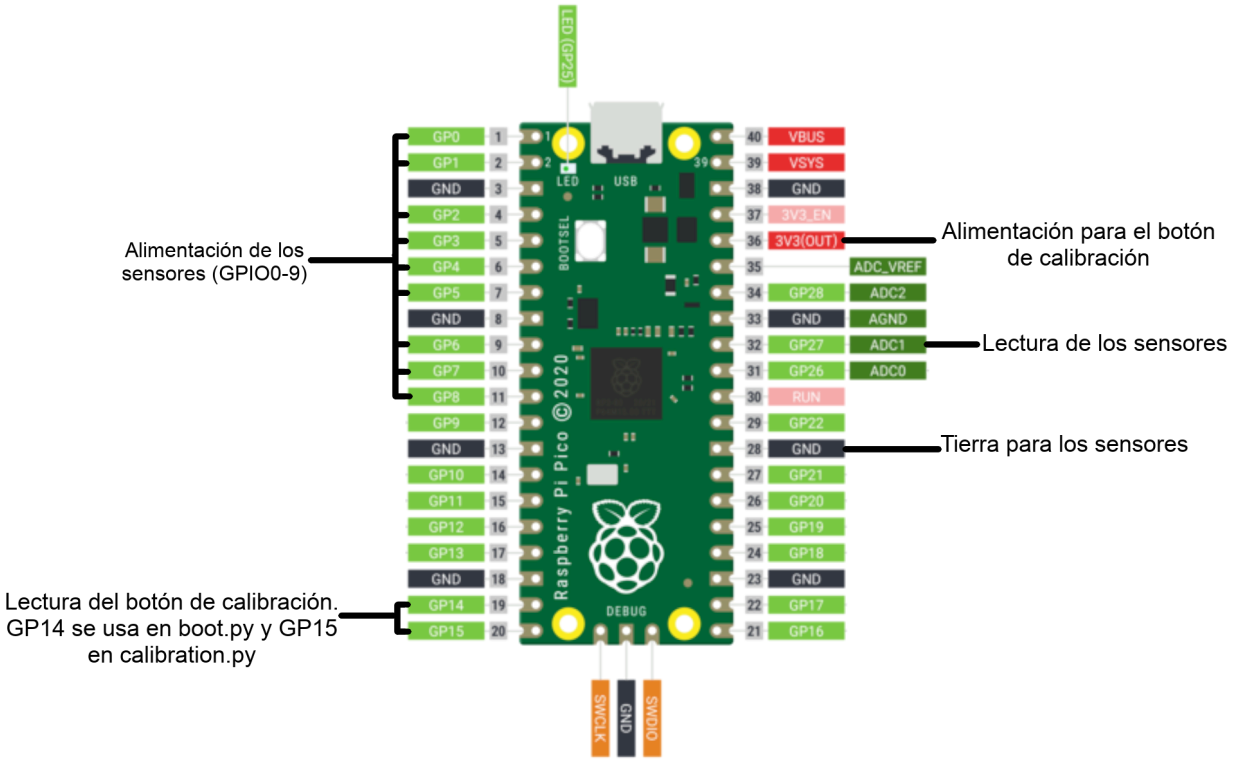
Teniendo ya un switch modificado, se probó cuánto era la variación máxima de voltaje causada por el movimiento del switch, la cual resultó ser de aproximadamente unos 120mV. Esta variación fue suficiente como para lograr la precisión que queríamos en el teclado. Como los sensores utilizados son lineales, para calcular la distancia del switch en milímetros se requiere solamente evaluar una regla de 3.

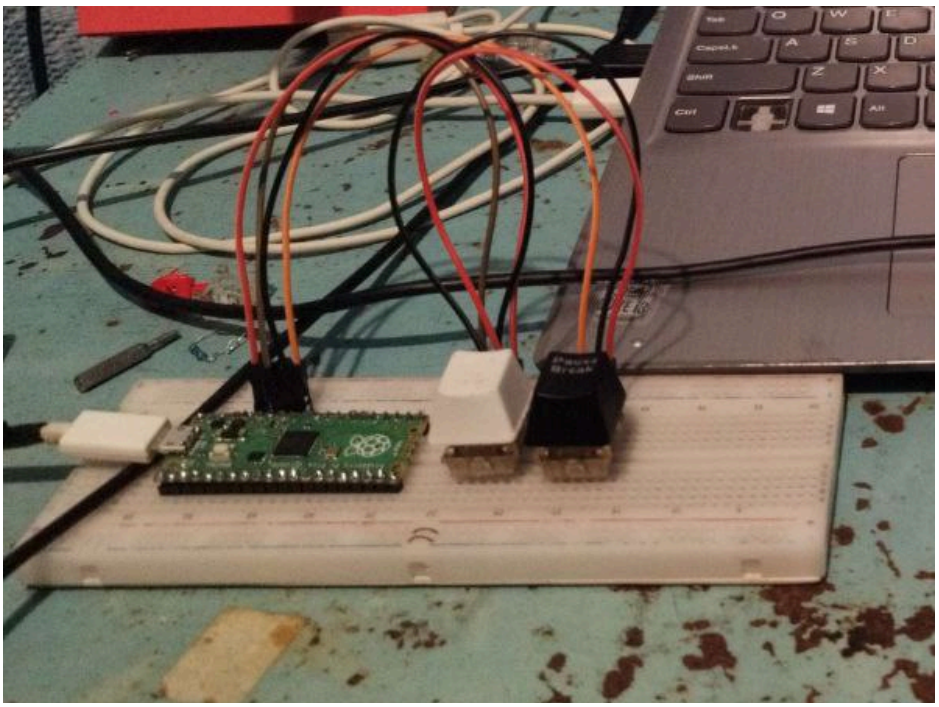
"La serie de productos OH490 son circuitos integrados de alta precisión, y el rango total de voltaje (Raíl-Raíl) de salida en el producto y efectivo para mediciones precisas. El voltaje de salida es definido por el voltaje de alimentación, y varía en proporción a la fuerza del campo magnético. Puede ser aplicado a medición de movimiento, distancia, sensor de posición y mucho más." (OUZHUO, n.d.).



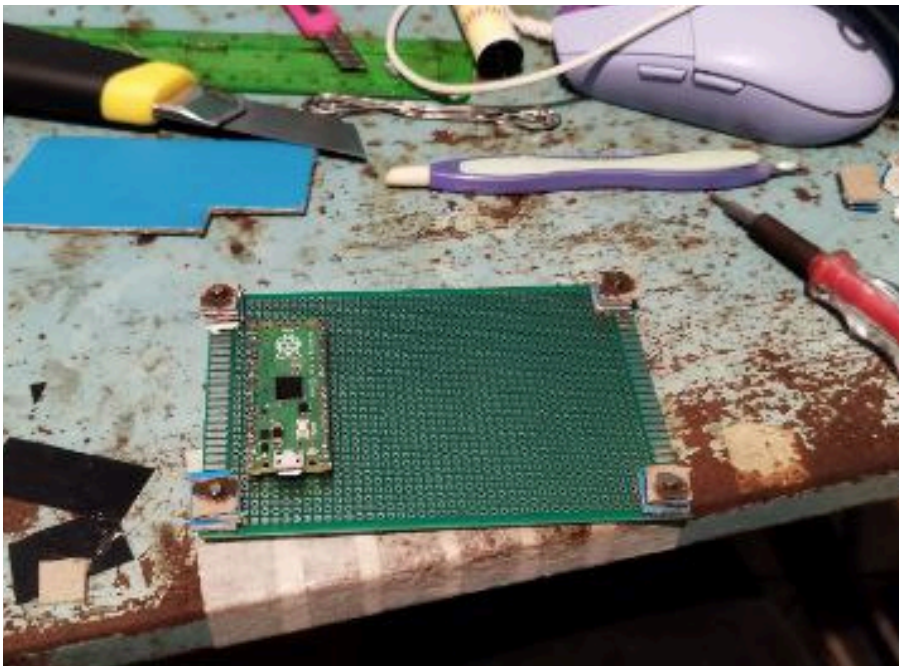
distancia = $(V_{actual} - V_{min}) * 4 / (V_{máx} - V_{min})$

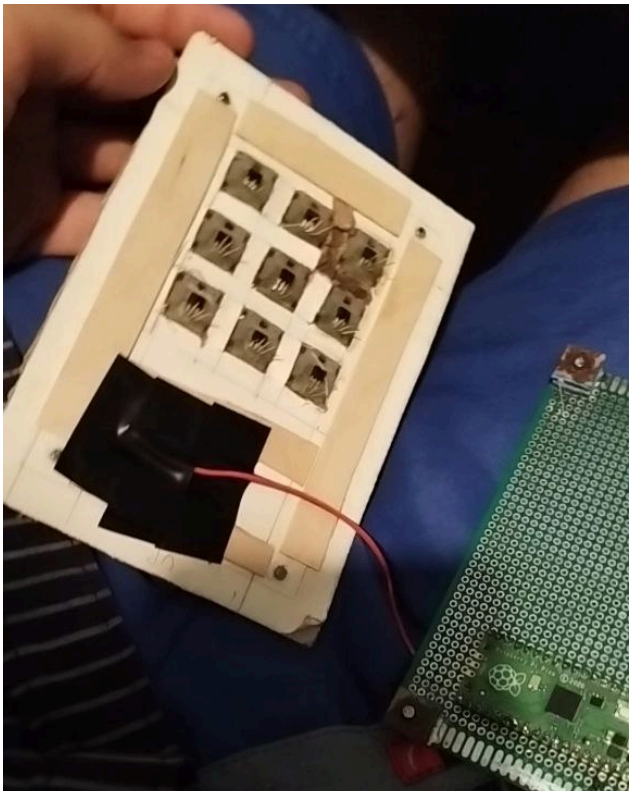
Se continuó modificando switches y probando la implementación planeada para leer todos los sensores desde un mismo ADC que consistía en alimentar un sensor a la vez para cada lectura, la cual resultó exitosa.

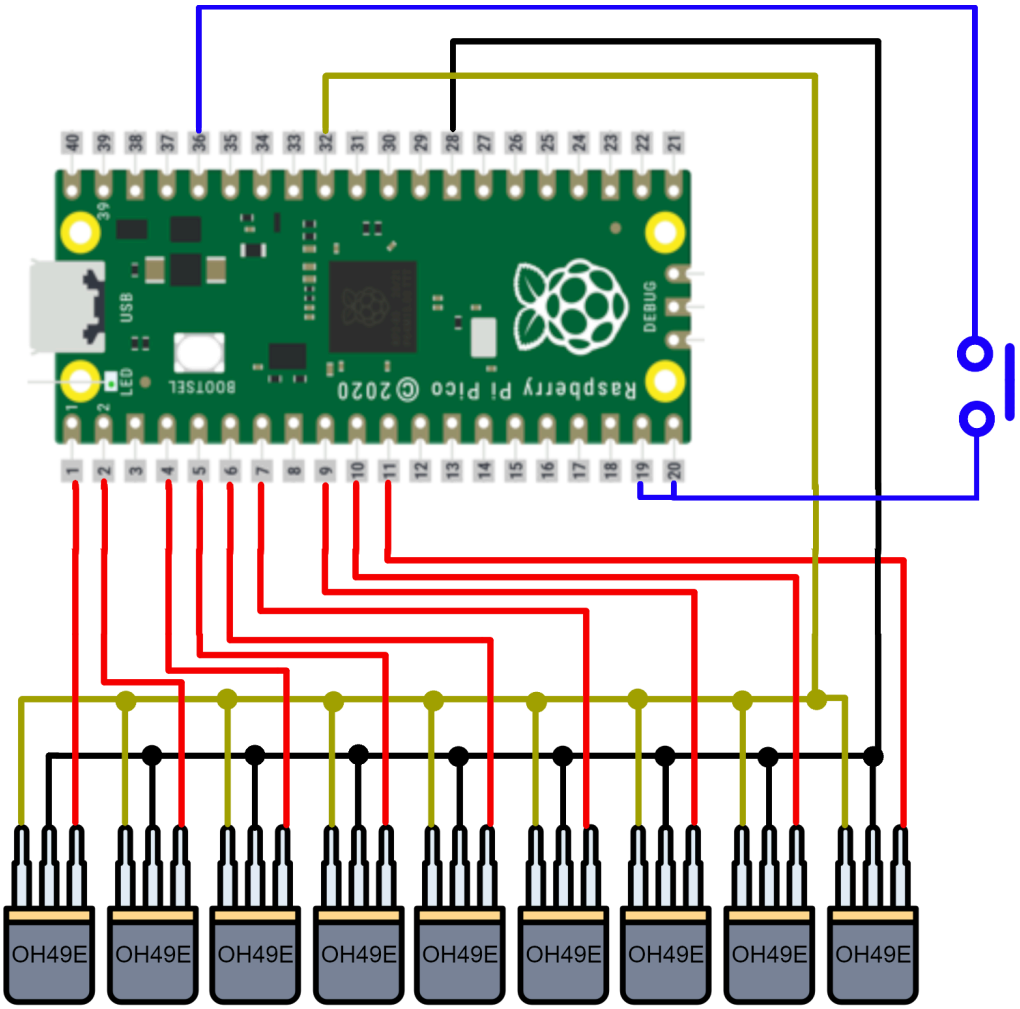




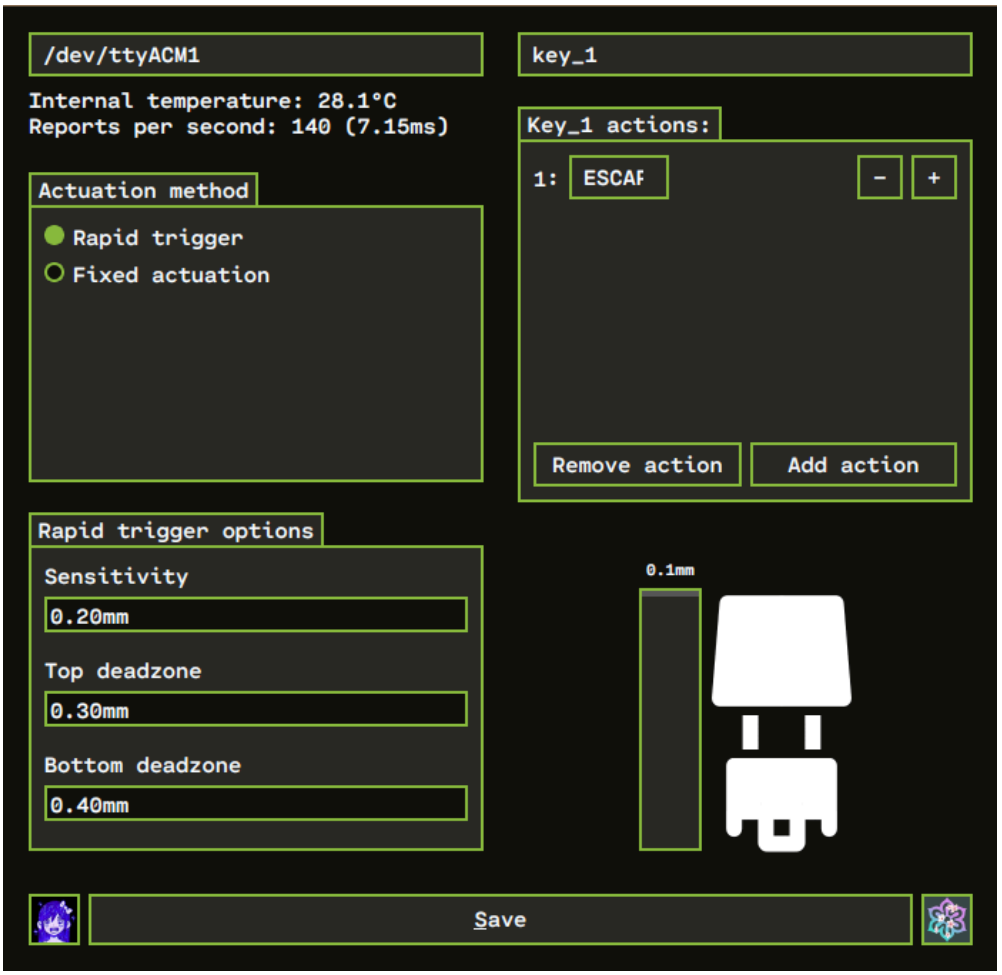
Habiendo probado que el código funcionaba con dos teclas, se procedió a hacer las nueve teclas totales y montar en una estructura hecha a base de cartón, foamy y paletas de madera.






OH49E
OH49E
OH49E
OH49E
OH49E
OH49E
OH49E
OH49E
OH49E

Con el teclado listo, se procedió a trabajar en una aplicación de escritorio escrita en python usando el framework Qt que nos permite ver el estado de cada una de las teclas así como configurar el kzooting.



La aplicación es capaz de enviar tres tipos de mensajes a través del puerto serial, a los cuales el kzooting responde adecuadamente:

- info_request: El kzooting envía un JSON con la información actual, la cual incluye la temperatura del microcontrolador, el tiempo de reporte,y la distancia y estado de cada una de las teclas.
- configs_request: El kzooting envía un JSON con sus configuraciones actuales. Este mensaje se envía cuando se selecciona un puerto serial para actualizar las configuraciones mostradas en la aplicación.

- <JSON de configuraciones>: Al apretar el botón “Save”, se envían las configuraciones modificadas en un JSON. El kzooting entonces reconfigura cada tecla adecuadamente y escribe la nueva configuración en el archivo “configs.json”.

Las posibles configuraciones a través de la app son las siguientes:

- Rapid trigger
    - Sensibilidad
    - Zona muerta superior
    - Zona muerta inferior
- Actuación fija
    - Punto de actuación
    - Reset

Desde la aplicación también se pueden programar las teclas y agregar macros. El mapeo de las teclas funciona de la siguiente manera: Cada tecla tiene acciones y cada acción tiene elementos llamados keycodes. Para cada acción, sus keycodes se mandan en orden, y se mantienen presionados hasta que termine la acción. Cuando se presiona una tecla, sus acciones se mandan en orden, con la diferencia de que todas los keycodes de la acción anterior han sido liberados. Es decir, para obtener una macro que envíe la string “Ab”, se necesita un mapeo como el siguiente:

Donde la “A” se envía con el “Shift” presionado, pero la “B” se envía de forma independiente. Si la tecla sólo contiene una acción, el envío de la tecla se efectúa como en cualquier teclado.

Dificultades

La mayor dificultad en cuanto a la parte del microcontrolador, fue el hecho de que necesitábamos leer nueve entradas analógicas con la Raspberry Pi Pico, que sólo cuenta con tres ADC. Para solucionar este problema tuvimos que rediseñar el montado original, en vez de alimentar todas las teclas a la vez y leer cada una en un ADC, montamos el circuito de forma que todos los sensores fueran al mismo ADC, pero fueran alimentados por pines digitales, de esta forma, para leer un sensor en específico ahora sólo tenemos que encender el pin que lo alimenta, leer el valor analógico, y luego apagar el pin para no afectar la siguiente lectura. Esta solución tuvo la consecuencia de agregar un ruido con un máximo de 0.1mm a cada lectura, el cual fue disminuido a aproximadamente 0.05mm cambiando el parámetro “avgrange” de 20 a 80 en las llamadas al método poll() lo cual resultó en valores menos volátiles.

Por la parte de la aplicación, tuvimos problemas al inicio debido a que empleamos un código con multithreading, lo cual complicó muchísimo la parte de la comunicación serial mediante comandos. El código de la aplicación se reestructuró para ejecutar todo en una sola thread y evitar race conditions.

Conclusiones

Al concluir este proyecto, logramos construir un teclado con entrada analógica con altas capacidades de personalización. El teclado a este punto ha sido activamente usado (principalmente para videojuegos) y ha presentado ventajas respecto a teclados convencionales, especialmente al hacer uso de la tecnología rapid trigger, la cual pudo ser replicada con exactitud luego de analizar detenidamente su funcionamiento en los teclados Wooting.

El código, tanto el utilizado en la pico como el de la aplicación de escritorio, fue estructurado de manera que permita fácilmente mapear las teclas con macros si así el usuario lo desea.

Código en el microcontrolador

boot.py

```
import supervisor
import storage
import board
import usb_cdc
import usb_hid
from digitalio import DigitalInOut, Direction, Pull

# Enable a data serial port
# Should change console to False
# if no more changes are to be made
usb_cdc.enable(console=True, data=True)

# Only enable keyboard
usb_hid.enable((usb_hid.Device.KEYBOARD,))

# Disable autoreloading jic
supervisor.runtime.autoreload = False

# Change volume name (in case it is gonna be mounted)
new_name = "KZOOTING"
storage.remount("/", readonly=False)
m = storage.getmount("/")
m.label = new_name
storage.remount("/", readonly=True)

# Calibration mode
calibration_button = DigitalInOut(board.GP15)
calibration_button.direction = Direction.INPUT
calibration_button.pull = Pull.DOWN

# Don't mount volume
#storage.disable_usb_drive()

if calibration_button.value:
    supervisor.set_next_code_file("calibration.py")
```

calibration.py

```
import json
import time
import board
import analogio
import keys
import supervisor
from digitalio import DigitalInOut, Direction, Pull


def calibrate():
    """
    Calibrate every key in the keys.key_list
    """
    # Set up button
    calibration_button = DigitalInOut(board.GP14)
    calibration_button.direction = Direction.INPUT
    calibration_button.pull = Pull.DOWN

    # Calibrate keys while button is held
    while calibration_button.value:
        for key in keys.key_list:
            key.poll(100)
            key.calibrate()

    # Log values
    calibration_dict = {}
    for key in keys.key_list:
        calibration_dict[key.id] = {}
        calibration_dict[key.id]["top_adc"] = key.top_adc
        calibration_dict[key.id]["bottom_adc"] = key.bottom_adc
    with open("calibration_values.json", "w") as calibration_file:
        json.dump(calibration_dict, calibration_file)

    # Start main once calibration is over
    supervisor.set_next_code_file("main.py")
    supervisor.reload()


if __name__ == "__main__":
    calibrate()
```

main.py

```
import usb_hid
import board
import json
import keys
import usb_cdc
import supervisor
from microcontroller import cpu
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keycode import Keycode


def setup_keys(configs):
    """
    Sets up the keys in the key_list
    using values from a json
    """
    general_configs = configs["general"]

    # Config keys
    for key in keys.key_list:

        # Set actions
        key_configs = configs[key.id]
        key_actions = key_configs["actions"]
        for i in range(len(key_actions)):
            try:
                for j in range(len(key_actions[i])):
                    key_actions[i][j] = getattr(Keycode,
key_actions[i][j])
                #key.actions.append(getattr(Keycode, action))
            except:
                pass
        key.actions = key_actions

        # Set everything else
        key.sensitivity = general_configs["sensitivity"]
        key.top_deadzone = general_configs["top_deadzone"]
        key.bottom_deadzone = general_configs["bottom_deadzone"]
        key.actuation_point = general_configs["actuation_point"]
        key.rapid_trigger = general_configs["rapid_trigger"]
```

```
counter = 0
def main():
    # SETUP
    # HID keyboard
    kbd = Keyboard(usb_hid.devices)

    # Serial in and out dictionary
    pico_info = {}
    pico_info["message_type"] = "info_request_response"

    # Load configs from file
    configs = ""
    with open("config.json", "r") as config_file:
        configs = json.load(config_file)

    setup_keys(configs)

    # Calibrate keys
    calibrations = ""
    with open("calibration_values.json", "r") as calibration_file:
        calibrations = json.load(calibration_file)

    for key in keys.key_list:
        # Set calibrations
        key_calibrations = calibrations[key.id]
        key.top_adc = key_calibrations["top_adc"]
        key.bottom_adc = key_calibrations["bottom_adc"]

        # Populate out dict
        pico_info[key.id] = {}

    # Prepare some info lists
    temperatures = []
    report_times = []
    smoothing_len = 20
    for _i in range(smoothing_len):
        temperatures.append(1)
        report_times.append(1)

    # LOOP
    counter = 0
```

```
    usb_cdc.data.flush()
    last_report = supervisor.ticks_ms()
    while True:
        # Get current report time
        report_times.append(supervisor.ticks_ms() - last_report)
        last_report = supervisor.ticks_ms()
        report_times.pop(0)

        # Get current temperature
        temperatures.append(cpu.temperature)
        temperatures.pop(0)

        # Average stuff
        report_times_avg = 0
        temperatures_avg = 0
        for i in range(smoothing_len):
            report_times_avg += report_times[i]
            temperatures_avg += temperatures[i]
        report_times_avg /= smoothing_len
        temperatures_avg /= smoothing_len

        # Log smooth stuff
        pico_info["temperature"] = temperatures_avg
        pico_info["report_time"] = report_times_avg

        for key in keys.key_list:
            key.poll(80)
            if key.rapid_trigger:
                key.evaluate_rapid_trigger()
            else:
                key.evaluate_fixed_actuation()
            if key.curr_state:
                if key.state_changed:
                    # Handle the keypress
                    # If action is a macro, just
                    # send it once
                    if len(key.actions) > 1:
                        for keycode in key.actions:
                            kbd.send(*keycode)
                    # If action is a single keycode list
                    # press and hold
                    else:
```

```
                        kbd.press(*key.actions[0])
                        pass
            else:
                if key.state_changed:
                    kbd.release(*key.actions[0])
                    pass
            pico_info[key.id]["state"] = key.curr_state
            pico_info[key.id]["distance"] = key.curr_dist

        # Read from serial port if data is available
        if usb_cdc.data.in_waiting > 0 and
usb_cdc.data.out_waiting == 0:
            in_data = usb_cdc.data.readline().decode()
            if in_data == "configs_request\n":
                out_dict = configs
                out_dict["message_type"] =
"configs_request_response"
                out_data = json.dumps(out_dict) + "\n"
                usb_cdc.data.write(out_data.encode())

            elif in_data == "info_request\n":
                out_dict = pico_info
                out_data = json.dumps(out_dict) + "\n"
                usb_cdc.data.write(out_data.encode())

            else: # This is a new config json
                # Set up keys again with the new info
                configs = json.loads(in_data)
                setup_keys(configs)

                # Write to configuration file
                with open("config.json", "w") as config_file:
                    json.dump(configs, config_file)
        counter += 1
if __name__ == "__main__":
    main()
```

keys.py

```
import analogio
import board
from digitalio import DigitalInOut, Direction, Pull


class Key():
    def __init__(self, id, adc, vcc):
        # Set up pins, id and evualuation method
        self.rapid_trigger = True
        self.id = id
        self.adc = adc
        self.vcc = DigitalInOut(vcc)
        self.vcc.direction = Direction.OUTPUT
        self.actions = []

        # Used for calibration
        self.bottom_adc = 0
        self.top_adc = 100000
        self.travel_dist = 4

        # Value that's updated for key presses
        self.curr_adc = 0
        self.curr_dist = 0

        # For rapid trigger
        self.sensitivity = 0.3
        self.top_deadzone = 1
        self.bottom_deadzone = 0.3
        self.hook = self.travel_dist - self.bottom_deadzone

        # For fixed actuation
        self.actuation_point = 1.5
        self.actuation_reset = 0.3

        # State of the switch
        self.curr_state = False
        self.state_changed = False


    def poll(self, avgrange):
        """
```

```
        Poll avgrange times and average values
        """
        # Turn on vcc for polling
        self.vcc.value = True

        avg = 0
        for _i in range(avgrange):
            avg += self.adc.value

        # Turn off vcc after polling
        self.vcc.value = False

        avg = avg/avgrange
        self.curr_adc = avg
        pass


    def adc_to_dist(self, adc):
        """
        Gets distance in mm from an adc value
        """
        # Normalize
        dist = (adc - self.top_adc)/(self.bottom_adc -
self.top_adc)

        # Make the function linear
        try:
            # 100% this doesn't work lmao, but it makes it better
            # dist = math.sqrt(dist)
            # Nvm, turns out it was already linear? idk anymore
            pass
        except:
            pass

        # Scale to travel distance
        dist = dist * self.travel_dist

        # Clamp for nicer values
        dist = max(min(self.travel_dist,dist),0)
        return dist
```

```
    def calibrate(self):
        """
        Calibrate key. Meant to be used repeatedly
        """
        # Calibrate values
        if self.curr_adc > self.bottom_adc:
            self.bottom_adc = (self.bottom_adc + self.curr_adc)/2
        elif self.curr_adc < self.top_adc:
            self.top_adc = (self.top_adc + self.curr_adc)/2

        # Failsafe
        if self.top_adc == self.bottom_adc:
            self.bottom_adc += 0.1
        pass


    def evaluate_rapid_trigger(self):
        """
        Wooting's rapid trigger technology
        """
        self.curr_dist = self.adc_to_dist(self.curr_adc)

        # Keep current distance and hook in a safe range
        if self.curr_dist > self.travel_dist -
self.bottom_deadzone:
            #self.curr_dist = self.travel_dist
            self.hook = self.travel_dist - self.bottom_deadzone
            self.update_state(True)
            return
        elif self.curr_dist < self.top_deadzone:
            #self.curr_dist = 0
            self.hook = self.top_deadzone
            self.update_state(False)
            return

        # Implement rapid trigger
        if self.curr_dist >=self.hook + self.sensitivity:
            self.hook = self.curr_dist
            self.update_state(True)
            return
        elif self.curr_dist <=self.hook - self.sensitivity:
            self.hook = self.curr_dist
```

```
            self.update_state(False)
            return
        self.update_state(self.curr_state)


    def evaluate_fixed_actuation(self):
        """
        Standard keyboard implementation
        """
        self.curr_dist = self.adc_to_dist(self.curr_adc)

        if self.curr_dist >=self.actuation_point:
            self.update_state(True)
            #self.curr_dist = self.travel_dist
            return
        elif self.curr_dist < self.actuation_point -
self.actuation_reset:
            self.update_state(False)
            #self.curr_dist = 0
            return
        self.update_state(self.curr_state)
        pass


    def update_state(self, state):
        """
        Updates key state and keeps track of whether the
        state of the key has changed or not
        """
        if state != self.curr_state:
            self.curr_state = state
            self.state_changed = True
        else:
            self.state_changed = False


# Keys in the keypad
key_adc = analogio.AnalogIn(board.GP27)
key_list = [Key(id="key_1",
                adc=key_adc,
                vcc=board.GP0),
            Key(id="key_2",
```

```
            adc=key_adc,
            vcc=board.GP1),
        Key(id="key_3",
            adc=key_adc,
            vcc=board.GP2),
        Key(id="key_4",
            adc=key_adc,
            vcc=board.GP3),
        Key(id="key_5",
            adc=key_adc,
            vcc=board.GP4),
        Key(id="key_6",
            adc=key_adc,
            vcc=board.GP5),
        Key(id="key_7",
            adc=key_adc,
            vcc=board.GP6),
        Key(id="key_8",
            adc=key_adc,
            vcc=board.GP7),
        Key(id="key_9",
            adc=key_adc,
            vcc=board.GP8)]
```

Código para la aplicación

kzooting_gui

```
#!/bin/python3

import kzserial
import os
import sys
import serial
import json
import keycodes
import collections
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import Qt, pyqtSignal, QObject, QTimer

PROGRAM_DIR = os.path.dirname(os.path.abspath(sys.argv[0]))
TOTAL_KEYS = 9


class State(QObject):
    """
    Implementation of the Observer Pattern to control application
    state in a centralized manner. Callbacks are always executed
    on the main thread of the application using Qt Signals.
    """

    __update_signal = pyqtSignal(str, object)

    def __init__(self):
        super(QObject, self).__init__()
        self.__dict__["listeners"] = collections.defaultdict(list)
        self.__update_signal.connect(self.__execute_callbacks)

    def attach_listener(self, property_name, callback):
        self.listeners[property_name].append(callback)

    def setter(self, property_name):
        return lambda value: self.__setattr__(property_name,
value)

    def __execute_callbacks(self, property_name, value):
```

```
        for callback in self.listeners.get(property_name, []):
            callback(value)

    def __setattr__(self, property_name, value):
        self.__dict__[property_name] = value
        self.__update_signal.emit(property_name, value)


class GeneralInfo(QWidget):
    """
    Top left quadrant of the window
    """

    def __init__(self, state):
        super(GeneralInfo, self).__init__()

        # Set up dropdown menu for ports
        ports_menu = QComboBox()

ports_menu.textActivated.connect(state.setter("selected_port"))
        self.was_empty = True

        def update_ports(ports):
            ports_menu.clear()
            for port in ports:
                ports_menu.addItem(port)
            if self.was_empty and len(ports):
                ports_menu.setCurrentText(ports[0])
                state.selected_port = ports[0]
                self.was_empty = False
            elif len(ports) == 0:
                self.was_empty = True

        state.attach_listener("available_ports", update_ports)

        # Set up info strings
        info_strings = QLabel()
        info_strings.setText(
            "Internal temperature: 0.00°C\n" "Reports per second:
000 (0.00ms)\n"
        )
        info_strings.setFixedWidth(300)
```

```
        state.attach_listener(
            "info",
            lambda info: info_strings.setText(
                f"Internal temperature:
{info['temperature']:.1f}°C\n"
                f"Reports per second:
{1000/info['report_time']:.0f} ({info['report_time']:.2f}ms)"
            ),
        )

        # Populate grid
        general_info_grid = QGridLayout()
        general_info_grid.addWidget(ports_menu, 0, 0, 1, 2)
        general_info_grid.addWidget(info_strings, 1, 0)
        self.setLayout(general_info_grid)


class GeneralConfigs(QWidget):
    """
    Bottom left quadrant of the window
    """

    def __init__(self, state):
        super(GeneralConfigs, self).__init__()
        self.state = state
        state.out_configs["general"] = {}

        # Add check boxes and input fields
        rapid_trigger_rb = QRadioButton("Rapid trigger")
        rapid_trigger_rb.setChecked(True)
        fixed_actuation_rb = QRadioButton("Fixed actuation")

        actuation_method_vbox = QVBoxLayout()
        actuation_method_vbox.addWidget(rapid_trigger_rb)
        actuation_method_vbox.addWidget(fixed_actuation_rb)
        actuation_method_vbox.addStretch()

        actuation_method = QGroupBox("Actuation method")
        actuation_method.setLayout(actuation_method_vbox)

        # Rapid trigger specific options
```

```
        sensitivity = QDoubleSpinBox()
        sensitivity.setSuffix("mm")
        sensitivity.setDecimals(2)
        sensitivity.setRange(0.1, 1)
        sensitivity.setSingleStep(0.05)
        sensitivity_label = QLabel()
        sensitivity_label.setText("Sensitivity")
        sensitivity_label.setBuddy(sensitivity)

        top_deadzone = QDoubleSpinBox()
        top_deadzone.setSuffix("mm")
        top_deadzone.setDecimals(2)
        top_deadzone.setRange(0.1, 1)
        top_deadzone.setSingleStep(0.05)
        top_deadzone_label = QLabel()
        top_deadzone_label.setText("Top deadzone")
        top_deadzone_label.setBuddy(top_deadzone)

        bottom_deadzone = QDoubleSpinBox()
        bottom_deadzone.setSuffix("mm")
        bottom_deadzone.setDecimals(2)
        bottom_deadzone.setRange(0.1, 1)
        bottom_deadzone.setSingleStep(0.05)
        bottom_deadzone_label = QLabel()
        bottom_deadzone_label.setText("Bottom deadzone")
        bottom_deadzone_label.setBuddy(bottom_deadzone)

        rt_options_vbox = QVBoxLayout()
        rt_options_vbox.addWidget(sensitivity_label)
        rt_options_vbox.addWidget(sensitivity)
        rt_options_vbox.addWidget(top_deadzone_label)
        rt_options_vbox.addWidget(top_deadzone)
        rt_options_vbox.addWidget(bottom_deadzone_label)
        rt_options_vbox.addWidget(bottom_deadzone)
        rt_options_vbox.addStretch()

        rt_options = QGroupBox("Rapid trigger options")
        rt_options.setLayout(rt_options_vbox)

        # Fixed actuation scpecific options
        actuation_point = QDoubleSpinBox()
        actuation_point.setSuffix("mm")
```

```
        actuation_point.setDecimals(2)
        actuation_point.setRange(0.5, 3.5)
        actuation_point.setSingleStep(0.05)
        actuation_point_label = QLabel()
        actuation_point_label.setText("Actuation point")
        actuation_point_label.setBuddy(actuation_point)

        actuation_reset = QDoubleSpinBox()
        actuation_reset.setSuffix("mm")
        actuation_reset.setDecimals(2)
        actuation_reset.setRange(0.1, 0.3)
        actuation_reset.setSingleStep(0.05)
        actuation_reset_label = QLabel()
        actuation_reset_label.setText("Actuation reset")
        actuation_reset_label.setBuddy(actuation_reset)

        fa_options_vbox = QVBoxLayout()
        fa_options_vbox.addWidget(actuation_point_label)
        fa_options_vbox.addWidget(actuation_point)
        fa_options_vbox.addWidget(actuation_reset_label)
        fa_options_vbox.addWidget(actuation_reset)

        fa_options = QGroupBox("Fixed actuation options")
        fa_options.setLayout(fa_options_vbox)
        fa_options_vbox.addStretch()

        # Populate grid
        sub_options = QStackedWidget()
        sub_options.addWidget(rt_options)
        sub_options.addWidget(fa_options)

        general_configs_grid = QGridLayout()
        general_configs_grid.addWidget(actuation_method, 0, 0)
        general_configs_grid.addWidget(sub_options, 1, 0)
        self.setLayout(general_configs_grid)

        def update_options_from_dict(configs):

sensitivity.setValue(configs["general"]["sensitivity"])

top_deadzone.setValue(configs["general"]["top_deadzone"])
```

```
bottom_deadzone.setValue(configs["general"]["bottom_deadzone"])

actuation_point.setValue(configs["general"]["actuation_point"])

actuation_reset.setValue(configs["general"]["actuation_reset"])

            # Update group box that is shown
            if configs["general"]["rapid_trigger"]:
                rapid_trigger_rb.setChecked(True)
                sub_options.setCurrentWidget(rt_options)
            else:
                fixed_actuation_rb.setChecked(True)
                sub_options.setCurrentWidget(fa_options)

        def update_dict_from_options():
            state.out_configs["general"]["rapid_trigger"] =
rapid_trigger_rb.isChecked()
            state.out_configs["general"]["sensitivity"] =
sensitivity.value()
            state.out_configs["general"]["top_deadzone"] =
top_deadzone.value()
            state.out_configs["general"]["bottom_deadzone"] =
bottom_deadzone.value()
            state.out_configs["general"]["actuation_point"] =
actuation_point.value()
            state.out_configs["general"]["actuation_reset"] =
actuation_reset.value()

            # Update group box that is shown
            if rapid_trigger_rb.isChecked():
                sub_options.setCurrentWidget(rt_options)
            else:
                sub_options.setCurrentWidget(fa_options)

        rapid_trigger_rb.toggled.connect(update_dict_from_options)
        sensitivity.valueChanged.connect(update_dict_from_options)

top_deadzone.valueChanged.connect(update_dict_from_options)

bottom_deadzone.valueChanged.connect(update_dict_from_options)

actuation_point.valueChanged.connect(update_dict_from_options)
```

```
actuation_reset.valueChanged.connect(update_dict_from_options)
        state.attach_listener("in_configs",
update_options_from_dict)


class RemapperComboBox(QComboBox):
    """
    The QComboBoxes shown in the remapper
    selection
    """

    def __init__(self, keycode):
        super(RemapperComboBox, self).__init__()

        for key_string in keycodes.values:
            self.addItem(key_string)
        self.setCurrentText(keycodes.strings[keycode])
        self.view().setMinimumWidth(170)
        # Alignment
        # self.setEditable(True)
        #
self.lineEdit().setAlignment(Qt.AlignmentFlag.AlignHCenter)
        # self.lineEdit().setReadOnly(True)


class RemapperHBox(QHBoxLayout):
    """
    The QHBoxLayout containing all keys
    of an action
    """

    def __init__(self, j, state, remap_sl, box_keycodes):
        super(RemapperHBox, self).__init__()

        current_combo_boxes = box_keycodes[state.selected_key]
        # Add label at the left
        number_label = QLabel(f"{j+1}:")
        number_label.setProperty("class", "ActionNumberLabel")
        self.addWidget(number_label)

        def update_dict_from_options():
```

```
            # Updates the out configs according to
            # the RemapperComboBoxes
            curr_key = int(state.selected_key[-1]) - 1

            # Get vbox
            curr_vbox = remap_sl.widget(curr_key).layout()

            # Get actions count, that is:
            # total items - stretch, buttons hbox (2)
            actions_count = curr_vbox.count() - 2
            for j in range(actions_count):
                # Get hbox
                curr_hbox = curr_vbox.itemAt(j).layout()

                # Get the keycode count. That is:
                # total items - label, stretch, buttons (4)
                keycode_count = curr_hbox.count() - 4

                # If there are less keycodes now, delete the old
                # remaining ones from the dict
                actions =
state.out_configs[state.selected_key]["actions"][j]
                actions_length = len(actions)
                while keycode_count < actions_length:
                    actions.pop(-1)

                # Change the dict's keycodes
                for k in range(keycode_count):
                    curr_combo_box = curr_hbox.itemAt(k +
1).widget()
                    curr_keycode =
keycodes.values[curr_combo_box.currentText()]
                    # If an index is already there, just change
the
                    # value
                    if k < actions_length:
                        actions[k] = curr_keycode
                    # Else, append it
                    else:
                        actions.append(curr_keycode)
```

```
state.out_configs[state.selected_key]["actions"][j] = actions

        # Add all combo boxes from the dict
        for k in range(len(current_combo_boxes[j])):
            new_remapper =
RemapperComboBox(current_combo_boxes[j][k])
            self.addWidget(new_remapper)

new_remapper.currentTextChanged.connect(update_dict_from_options)

        self.addStretch()

        def add_key():
            # Get items
            item_count = self.count()
            add_button_item = self.takeAt(item_count - 1)
            remove_button_item = self.takeAt(item_count - 2)
            stretch_item = self.takeAt(item_count - 3)

            # Remove buttons and stretch from layout
            self.removeItem(add_button_item)
            self.removeItem(remove_button_item)
            self.removeItem(stretch_item)

            # Add the new RemapperComboBox
            new_remapper = RemapperComboBox(4)
            self.addWidget(new_remapper)

new_remapper.currentTextChanged.connect(update_dict_from_options)

            # Add buttons and stretch again
            self.addItem(stretch_item)
            self.addItem(remove_button_item)
            self.addItem(add_button_item)

            # Reflect the changes made in
            # the box_keycodes dict
            box_keycodes[state.selected_key][j].append(4)

        def remove_key():
            # Only remove if there is at least 2
            # keys
```

```
            item_count = self.count()
            if item_count < 6:
                return

            # Remove the last key from the layout
            last_key = self.takeAt(item_count - 4)
            self.removeItem(last_key)
            last_key_widget = last_key.widget()
            last_key_widget.deleteLater()

            # Reflect the changes made in
            # the box_keycodes dict
            box_keycodes[state.selected_key][j].pop(-1)
            update_dict_from_options()

        # Add buttons at the right
        remove_button = QPushButton("-")
        remove_button.clicked.connect(remove_key)
        self.addWidget(remove_button)

        add_button = QPushButton("+")
        add_button.clicked.connect(add_key)
        self.addWidget(add_button)


class KeyConfigs(QWidget):
    """
    Top right quadrant of the window
    """

    def __init__(self, state):
        super(KeyConfigs, self).__init__()
        state.selected_key = "key_1"

        # Set up dropdown menu for keys
        keys_menu = QComboBox()
        keys_menu.setProperty("class", "KeysComboBox")

        for i in range(TOTAL_KEYS):
            state.out_configs[f"key_{i+1}"] = {}
            keys_menu.addItem(f"key_{i+1}")
```

```
        # Remapper
        box_keycodes = {"key_1": [[4]]}
        remap_sl = QStackedWidget()
        for i in range(TOTAL_KEYS):
            remap_gb = QGroupBox(f"Key_{i+1} actions:")
            remap_gb.setProperty("class", "RemapperGroupBox")
            remap_gb.setLayout(QVBoxLayout())
            remap_sl.addWidget(remap_gb)

        # Populate grid
        key_configs_grid = QGridLayout()
        key_configs_grid.addWidget(keys_menu, 0, 0)
        key_configs_grid.addWidget(remap_sl, 1, 0)
        self.setLayout(key_configs_grid)

        def update_keycodes(configs):
            # This function wouldn't be here if
            # the keycodes were the value held by
            # "key_id" keys, but I chose to put them
            # inside another key called "actions" in case
            # I ever wanna add per key configurations
            state.out_configs = configs
            for i in range(TOTAL_KEYS):
                actions = configs[f"key_{i+1}"]["actions"]
                box_keycodes[f"key_{i+1}"] = actions

            update_options_from_key(int(state.selected_key[-1]) -
1)

        def empty_layout(layout):
            # Empties a layout recursively
            while layout.count():
                to_delete = layout.itemAt(0)
                if to_delete is not None:
                    child_layout = to_delete.layout()
                    if child_layout is not None:
                        empty_layout(child_layout)
                    child_widget = to_delete.widget()
                    if child_widget is not None:
                        child_widget.deleteLater()
                    layout.removeItem(to_delete)
```

```
        def update_options_from_key(key_index):
            key_string = f"key_{key_index+1}"
            state.setter("selected_key")(key_string)
            remap_sl.setCurrentIndex(key_index)
            group_box = remap_sl.currentWidget()
            actions_vbox = group_box.layout()
            current_combo_boxes = box_keycodes[key_string]

            # First delete all previous widgets
            empty_layout(actions_vbox)

            # Then populate the grid again
            for j in range(len(current_combo_boxes)):
                keys_hbox = RemapperHBox(j, state, remap_sl,
box_keycodes)
                actions_vbox.addLayout(keys_hbox)
            actions_vbox.addStretch()

            def add_action():
                # Update box_keycodes dict
                box_keycodes[state.selected_key].append([4])

                # Get items
                item_count = actions_vbox.count()
                buttons_item = actions_vbox.takeAt(item_count - 1)
                stretch_item = actions_vbox.takeAt(item_count - 2)

                # Remove buttons from layout
                actions_vbox.removeItem(buttons_item)
                actions_vbox.removeItem(stretch_item)

                # Add the new RemapperHBox
                actions_vbox.addLayout(
                    RemapperHBox(item_count - 2, state, remap_sl,
box_keycodes)
                )

                # Add buttons and stretch again
                actions_vbox.addItem(stretch_item)
                actions_vbox.addItem(buttons_item)

            def remove_action():
```

```
                # Only remove if there are at least
                # two actions
                item_count = actions_vbox.count()
                if item_count < 4:
                    return

                # Remove the last action from the layout
                last_action = actions_vbox.takeAt(item_count - 3)
                last_action_layout = last_action.layout()
                empty_layout(last_action_layout)
                actions_vbox.removeItem(last_action)

                # Reflect the changes on the
                # box_keycodes dict
                box_keycodes[state.selected_key].pop(-1)

            # Now add buttons
            buttons_hbox = QHBoxLayout()
            add_button = QPushButton("Add action")
            add_button.clicked.connect(add_action)
            remove_button = QPushButton("Remove action")
            remove_button.clicked.connect(remove_action)
            buttons_hbox.addWidget(remove_button)
            buttons_hbox.addWidget(add_button)
            actions_vbox.addLayout(buttons_hbox)


keys_menu.currentIndexChanged.connect(update_options_from_key)

        state.attach_listener("in_configs", update_keycodes)


class Visualizer(QWidget):
    """
    Bottom right quadrant of the window
    """

    def __init__(self, state):
        super(Visualizer, self).__init__()

        # Set up the progress bar
        bar = QProgressBar()
```

```
        bar.setTextVisible(False)
        bar.setOrientation(Qt.Vertical)
        bar.setMaximum(400)
        bar.setMinimum(0)
        bar.setInvertedAppearance(True)
        bar_label = QLabel()
        bar_label.setProperty("class", "BarLabel")
        bar_label.setText("0.0 mm")
        bar_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        bar_label.setBuddy(bar)
        bar_vbox = QVBoxLayout()
        bar_vbox.addWidget(bar_label)
        bar_vbox.addWidget(bar)

        # Set up switch icon
        # The switch is divided in 3 to create an
        # up and down motion
        switch_top = QPixmap(PROGRAM_DIR +
"/../assets/switch_top.png").scaledToWidth(
            100, Qt.SmoothTransformation
        )
        switch_top_label = QLabel()
        switch_top_label.setPixmap(switch_top)

        switch_mid = QPixmap(PROGRAM_DIR +
"/../assets/switch_mid.png").scaledToWidth(
            100, Qt.SmoothTransformation
        )
        switch_mid_label = QLabel()
        switch_mid_label.setPixmap(switch_mid)
        switch_mid_height = 25
        switch_mid_label.setFixedHeight(switch_mid_height)

        switch_bottom = QPixmap(
            PROGRAM_DIR + "/../assets/switch_bottom.png"
        ).scaledToWidth(100, Qt.SmoothTransformation)
        switch_bottom_label = QLabel()
        switch_bottom_label.setPixmap(switch_bottom)
        switch_bottom_label.setProperty("class",
"SwitchBottomLabel")

        switch_vbox = QVBoxLayout()
```

```
        switch_vbox.addStretch()
        switch_vbox.addWidget(switch_top_label)
        switch_vbox.addWidget(switch_mid_label)
        switch_vbox.addWidget(switch_bottom_label)

        # Populate grid
        hbox = QHBoxLayout()
        hbox.addStretch()
        hbox.addLayout(bar_vbox)
        hbox.addLayout(switch_vbox)
        self.setLayout(hbox)
        hbox.addStretch()

        def update_bar_contents(info):
            distance = info[state.selected_key]["distance"]
            bar.setValue(round(distance * 100))
            bar_label.setText("{:.1f}".format(distance) + "mm")
            switch_mid_label.setFixedHeight(
                round(switch_mid_height - switch_mid_height *
distance / 4)
            )

            # Update color
            if info[state.selected_key]["state"]:
                bar.setProperty("class", "ActiveBar")
            else:
                bar.setProperty("class", "InactiveBar")

            # Make the bar update its style
            bar.style().polish(bar)

        state.attach_listener("info", update_bar_contents)


class MainWindow(QMainWindow):
    """
    Main aplication window
    """

    def __init__(self):
        super(MainWindow, self).__init__()
        state = State()
```

```
        self.state = state
        state.in_configs = {}
        state.out_configs = {}

        # Use local qss
        self.setStyleSheet(open(PROGRAM_DIR + "/style.qss",
"r").read())

        # Add a grid
        self.root_grid = QGridLayout()
        self.root_window = QWidget()
        self.root_window.setLayout(self.root_grid)
        self.setCentralWidget(self.root_window)

        # Create widgets
        general_info = GeneralInfo(state)
        general_configs = GeneralConfigs(state)
        key_configs = KeyConfigs(state)
        visualizer = Visualizer(state)

        # Lower side
        save_button = QPushButton("&Save", self)
        save_button.setProperty("class", "SaveButton")
        save_button.clicked.connect(self.send_configs_to_pico)
        kz_icon = QPixmap(PROGRAM_DIR +
"/../assets/kz_icon.jpeg").scaled(
            32, 32, Qt.IgnoreAspectRatio, Qt.SmoothTransformation
        )
        kz_label = QLabel()
        kz_label.setProperty("class", "KzLabel")
        kz_label.setFixedWidth(45)
        kz_label.setPixmap(kz_icon)

        degen_icon = QPixmap(PROGRAM_DIR +
"/../assets/degen_icon.png").scaled(
            32, 32, Qt.IgnoreAspectRatio, Qt.SmoothTransformation
        )
        degen_label = QLabel()
        degen_label.setProperty("class", "DegenLabel")
        degen_label.setFixedWidth(45)
        degen_label.setPixmap(degen_icon)
```

```
        lower_side = QHBoxLayout()
        lower_side.addWidget(kz_label)
        lower_side.addWidget(save_button)
        lower_side.addWidget(degen_label)

        # Left side
        left_side = QVBoxLayout()
        left_side.addWidget(general_info)
        left_side.addWidget(general_configs)

        # Right side
        right_side = QVBoxLayout()
        right_side.addWidget(key_configs)
        right_side.addWidget(visualizer)

        # Populate grid
        self.root_grid.addLayout(left_side, 0, 0)
        self.root_grid.addLayout(right_side, 0, 1)
        self.root_grid.addLayout(lower_side, 2, 0, 1, 2)

        # Watch for changes in ports directory
        self.rpp = None

        def update_ports_list():
            """
            Updates available ports
            """
            new_ports = kzserial.get_serial_ports()
            if new_ports != self.state.available_ports:
                self.state.available_ports =
kzserial.get_serial_ports()

        state.attach_listener("selected_port", lambda port:
self.update_open_port(port))

        self.state.available_ports = kzserial.get_serial_ports()
        timer1 = QTimer(self)
        timer1.timeout.connect(update_ports_list)
        timer1.start(500)

        timer2 = QTimer(self)
        timer2.timeout.connect(self.get_info_from_pico)
```

```
        timer2.start(16)

    def send_configs_to_pico(self):
        """
        Sends a configs to the pico to be rewritten
        """
        try:
            configs_dict = self.state.out_configs
            configs_json = json.dumps(configs_dict)
            self.rpp.write((configs_json + "\n").encode())
        except (OSError, serial.SerialException,
json.JSONDecodeError):
            pass

    def update_open_port(self, port):
        """
        Opens a new port and closes the last one
        """
        try:
            if self.rpp != None:
                self.rpp.close()
            self.rpp = serial.Serial(port, timeout=1)
            response =
kzserial.get_response_from_request(self.rpp, "configs_request")
            self.state.in_configs = response
        except (OSError, serial.SerialException,
json.JSONDecodeError) as e:
            self.rpp = None

    def get_info_from_pico(self):
        """
        Updates general information
        """
        try:
            if self.rpp != None:
                response =
kzserial.get_response_from_request(self.rpp, "info_request")
                self.state.info = response
        except (OSError, serial.SerialException,
json.JSONDecodeError) as e:
            pass
```

```
    def closeEvent(self, a0: QCloseEvent):
        return super().closeEvent(a0)


def main():
    app = QApplication(sys.argv)
    main_window = MainWindow()
    main_window.setWindowTitle("kzooting - GUI")
    main_window.show()
    sys.exit(app.exec())


if __name__ == "__main__":
    main()
```

kzserial.py

```
import serial
import sys
import glob
import json


def get_serial_ports():
    """
    Returns a list of all serial  ports
    that *can* be open
    """
    if sys.platform.startswith("win"):
        ports = ["COM%s" % (i + 1) for i in range(256)]
    elif sys.platform.startswith("linux") or
sys.platform.startswith("cygwin"):
        # this excludes your current terminal "/dev/tty"
        ports = glob.glob("/dev/tty[A-Za-z]*")
    elif sys.platform.startswith("darwin"):
        ports = glob.glob("/dev/tty.*")
    else:
        raise EnvironmentError("Unsupported platform")

    result = []
    for port in ports:
        try:
            s = serial.Serial(port)
            s.close()
            result.append(port)
        except (OSError, serial.SerialException):
            pass

    return result


def read_dict_from_port(port):
    """
    Reads a dictionary from the port provided
    """
    line = port.readline().decode()
    return json.loads(line)
```

```
def get_response_from_request(port, request):
    """
    Sends a request to the pico and waits until
    it responds to that request with an
    adequate message
    """
    port.write((request + "\n").encode())
    data = read_dict_from_port(port)
    return data
```

keycodes.py

```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft for Adafruit
Industries
#
# SPDX-License-Identifier: MIT

"""
`adafruit_hid.keycode.Keycode`
====================================================

* Author(s): Scott Shawcroft, Dan Halbert
"""


class Keycode:
    """USB HID Keycode constants.

    This list is modeled after the names for USB keycodes defined
in
    https://usb.org/sites/default/files/hut1_21_0.pdf#page=83.
    This list does not include every single code, but does include
all the keys on
    a regular PC or Mac keyboard.

    Remember that keycodes are the names for key *positions* on a
US keyboard, and may
    not correspond to the character that you mean to send if you
want to emulate non-US keyboard.
    For instance, on a French keyboard (AZERTY instead of QWERTY),
    the keycode for 'q' is used to indicate an 'a'. Likewise, 'y'
represents 'z' on
    a German keyboard. This is historical: the idea was that the
keycaps could be changed
    without changing the keycodes sent, so that different firmware
was not needed for
    different variations of a keyboard.
    """

    # pylint: disable-msg=invalid-name
    A = 0x04
    """``a`` and ``A``"""
    B = 0x05
```

```
    """``b`` and ``B``"""
    C = 0x06
    """``c`` and ``C``"""
    D = 0x07
    """``d`` and ``D``"""
    E = 0x08
    """``e`` and ``E``"""
    F = 0x09
    """``f`` and ``F``"""
    G = 0x0A
    """``g`` and ``G``"""
    H = 0x0B
    """``h`` and ``H``"""
    I = 0x0C
    """``i`` and ``I``"""
    J = 0x0D
    """``j`` and ``J``"""
    K = 0x0E
    """``k`` and ``K``"""
    L = 0x0F
    """``l`` and ``L``"""
    M = 0x10
    """``m`` and ``M``"""
    N = 0x11
    """``n`` and ``N``"""
    O = 0x12
    """``o`` and ``O``"""
    P = 0x13
    """``p`` and ``P``"""
    Q = 0x14
    """``q`` and ``Q``"""
    R = 0x15
    """``r`` and ``R``"""
    S = 0x16
    """``s`` and ``S``"""
    T = 0x17
    """``t`` and ``T``"""
    U = 0x18
    """``u`` and ``U``"""
    V = 0x19
    """``v`` and ``V``"""
    W = 0x1A
```

```
    """``w`` and ``W``"""
    X = 0x1B
    """``x`` and ``X``"""
    Y = 0x1C
    """``y`` and ``Y``"""
    Z = 0x1D
    """``z`` and ``Z``"""

    ONE = 0x1E
    """``1`` and ``!``"""
    TWO = 0x1F
    """``2`` and ``@``"""
    THREE = 0x20
    """``3`` and ``#``"""
    FOUR = 0x21
    """``4`` and ``$``"""
    FIVE = 0x22
    """``5`` and ``%``"""
    SIX = 0x23
    """``6`` and ``^``"""
    SEVEN = 0x24
    """``7`` and ``&``"""
    EIGHT = 0x25
    """``8`` and ``*``"""
    NINE = 0x26
    """``9`` and ``(``"""
    ZERO = 0x27
    """``0`` and ``)``"""
    ENTER = 0x28
    """Enter (Return)"""
    RETURN = ENTER
    """Alias for ``ENTER``"""
    ESCAPE = 0x29
    """Escape"""
    BACKSPACE = 0x2A
    """Delete backward (Backspace)"""
    TAB = 0x2B
    """Tab and Backtab"""
    SPACEBAR = 0x2C
    """Spacebar"""
    SPACE = SPACEBAR
    """Alias for SPACEBAR"""
```

```
    MINUS = 0x2D
    """``-`` and ``_``"""
    EQUALS = 0x2E
    """``=`` and ``+``"""
    LEFT_BRACKET = 0x2F
    """``[`` and ``{``"""
    RIGHT_BRACKET = 0x30
    """``]`` and ``}``"""
    BACKSLASH = 0x31
    r"""``\`` and ``|``"""
    POUND = 0x32
    """``#`` and ``~`` (Non-US keyboard)"""
    SEMICOLON = 0x33
    """``;`` and ``:``"""
    QUOTE = 0x34
    """``'`` and ``"``"""
    GRAVE_ACCENT = 0x35
    r""":literal:`\`` and ``~``"""
    COMMA = 0x36
    """``,`` and ``<``"""
    PERIOD = 0x37
    """``.`` and ``>``"""
    FORWARD_SLASH = 0x38
    """``/`` and ``?``"""

    CAPS_LOCK = 0x39
    """Caps Lock"""

    F1 = 0x3A
    """Function key F1"""
    F2 = 0x3B
    """Function key F2"""
    F3 = 0x3C
    """Function key F3"""
    F4 = 0x3D
    """Function key F4"""
    F5 = 0x3E
    """Function key F5"""
    F6 = 0x3F
    """Function key F6"""
    F7 = 0x40
    """Function key F7"""
```

```
    F8 = 0x41
    """Function key F8"""
    F9 = 0x42
    """Function key F9"""
    F10 = 0x43
    """Function key F10"""
    F11 = 0x44
    """Function key F11"""
    F12 = 0x45
    """Function key F12"""

    PRINT_SCREEN = 0x46
    """Print Screen (SysRq)"""
    SCROLL_LOCK = 0x47
    """Scroll Lock"""
    PAUSE = 0x48
    """Pause (Break)"""

    INSERT = 0x49
    """Insert"""
    HOME = 0x4A
    """Home (often moves to beginning of line)"""
    PAGE_UP = 0x4B
    """Go back one page"""
    DELETE = 0x4C
    """Delete forward"""
    END = 0x4D
    """End (often moves to end of line)"""
    PAGE_DOWN = 0x4E
    """Go forward one page"""

    RIGHT_ARROW = 0x4F
    """Move the cursor right"""
    LEFT_ARROW = 0x50
    """Move the cursor left"""
    DOWN_ARROW = 0x51
    """Move the cursor down"""
    UP_ARROW = 0x52
    """Move the cursor up"""

    KEYPAD_NUMLOCK = 0x53
    """Num Lock (Clear on Mac)"""
```

```
    KEYPAD_FORWARD_SLASH = 0x54
    """Keypad ``/``"""
    KEYPAD_ASTERISK = 0x55
    """Keypad ``*``"""
    KEYPAD_MINUS = 0x56
    """Keyapd ``-``"""
    KEYPAD_PLUS = 0x57
    """Keypad ``+``"""
    KEYPAD_ENTER = 0x58
    """Keypad Enter"""
    KEYPAD_ONE = 0x59
    """Keypad ``1`` and End"""
    KEYPAD_TWO = 0x5A
    """Keypad ``2`` and Down Arrow"""
    KEYPAD_THREE = 0x5B
    """Keypad ``3`` and PgDn"""
    KEYPAD_FOUR = 0x5C
    """Keypad ``4`` and Left Arrow"""
    KEYPAD_FIVE = 0x5D
    """Keypad ``5``"""
    KEYPAD_SIX = 0x5E
    """Keypad ``6`` and Right Arrow"""
    KEYPAD_SEVEN = 0x5F
    """Keypad ``7`` and Home"""
    KEYPAD_EIGHT = 0x60
    """Keypad ``8`` and Up Arrow"""
    KEYPAD_NINE = 0x61
    """Keypad ``9`` and PgUp"""
    KEYPAD_ZERO = 0x62
    """Keypad ``0`` and Ins"""
    KEYPAD_PERIOD = 0x63
    """Keypad ``.`` and Del"""
    KEYPAD_BACKSLASH = 0x64
    """Keypad ``\\`` and ``|`` (Non-US)"""

    APPLICATION = 0x65
    """Application: also known as the Menu key (Windows)"""
    POWER = 0x66
    """Power (Mac)"""
    KEYPAD_EQUALS = 0x67
    """Keypad ``=`` (Mac)"""
    F13 = 0x68
```

```
    """Function key F13 (Mac)"""
    F14 = 0x69
    """Function key F14 (Mac)"""
    F15 = 0x6A
    """Function key F15 (Mac)"""
    F16 = 0x6B
    """Function key F16 (Mac)"""
    F17 = 0x6C
    """Function key F17 (Mac)"""
    F18 = 0x6D
    """Function key F18 (Mac)"""
    F19 = 0x6E
    """Function key F19 (Mac)"""

    F20 = 0x6F
    """Function key F20"""
    F21 = 0x70
    """Function key F21"""
    F22 = 0x71
    """Function key F22"""
    F23 = 0x72
    """Function key F23"""
    F24 = 0x73
    """Function key F24"""

    LEFT_CONTROL = 0xE0
    """Control modifier left of the spacebar"""
    CONTROL = LEFT_CONTROL
    """Alias for LEFT_CONTROL"""
    LEFT_SHIFT = 0xE1
    """Shift modifier left of the spacebar"""
    SHIFT = LEFT_SHIFT
    """Alias for LEFT_SHIFT"""
    LEFT_ALT = 0xE2
    """Alt modifier left of the spacebar"""
    ALT = LEFT_ALT
    """Alias for LEFT_ALT; Alt is also known as Option (Mac)"""
    OPTION = ALT
    """Labeled as Option on some Mac keyboards"""
    LEFT_GUI = 0xE3
    """GUI modifier left of the spacebar"""
    GUI = LEFT_GUI
```

```
    """Alias for LEFT_GUI; GUI is also known as the Windows key,
Command (Mac), or Meta"""
    WINDOWS = GUI
    """Labeled with a Windows logo on Windows keyboards"""
    COMMAND = GUI
    """Labeled as Command on Mac keyboards, with a clover glyph"""
    RIGHT_CONTROL = 0xE4
    """Control modifier right of the spacebar"""
    RIGHT_SHIFT = 0xE5
    """Shift modifier right of the spacebar"""
    RIGHT_ALT = 0xE6
    """Alt modifier right of the spacebar"""
    RIGHT_GUI = 0xE7
    """GUI modifier right of the spacebar"""

# Keycode - String dicts
strings = {}
values = {}
for keycode_string in Keycode.__dict__:
    if not "__" in keycode_string:
        keycode = Keycode.__dict__[keycode_string]
        values[keycode_string] = keycode
        strings[keycode] = keycode_string
```

Referencias

Dexerto. (2022, August 8). *Wooting 60HE review: The fastest gaming keyboard*. Dexerto. Recuperado el 7 de Julio, 2023, de https://www.dexerto.com/tech/wooting-60he-review-1896356/

OUZHUO. (n.d.). Linear Hall IC. Recuperado el 7 de Julio, 2023, de https://ohhallsensor.com/product/oh495a-oh496b-linear-hall-ic/

Wooting. (n.d.). Wooting keyboards. Recuperado el 7 de Julio, 2023, de https://wooting.io/