



Escuela Técnica
Superior
de Ingeniería de
Telecomunicación



Universidad Politécnica de Cartagena

Desarrollo de un teclado para el S.XXI, personalizable y con QMK

Autor: Pablo Martínez Bernal

Director: José Alfonso Vera Repullo

Máster Universitario en Ingeniería de Telecomunicación

Curso 2022-23



Universidad
Politécnica
de Cartagena

RESUMEN

En la actualidad es cada vez más la gente que pasa varias horas al día delante de un ordenador, nuestra principal herramienta para controlarlos es el teclado y, sin embargo, su diseño apenas ha avanzado desde su aparición. Este diseño arcaico supone problemas de salud, falta de accesibilidad y reduce la productividad.

Por esto, en el presente documento se estudian los avances y variaciones que ha realizado la comunidad de aficionados en los últimos años y se detalla el diseño y construcción de un teclado más acorde al mundo actual que solventa los problemas recién citados, así como permitiendo una gran capacidad de personalización gracias a su diseño hardware que permite cambiar componentes fácilmente y el desarrollo de un firmware y software *open source* fácilmente editables.

ÍNDICE

1. Introducción	3
1.1. Motivación	3
1.2. Objetivo	3
2. Preámbulo	4
2.1. Requisitos previos	4
2.2. Configuración para trabajar en remoto	4
2.2.1. SSH	4
2.2.2. Montar disco en red	4
2.2.3. git	5
2.2.4. Servidor X	5
2.2.5. L ^A T _E X	5
3. Estado del arte	7
3.1. Diseño	7
3.1.1. Diseño anticuado	7
3.1.2. Algunos avances	9
3.2. Firmware	11
3.3. Hardware	11
4. Diseño del hardware	12
4.1. Cableado de las teclas	12
4.1.1. Conexión directa	12
4.1.2. Matriz	12
5. Desarrollo del firmware	14
5.1. Instalación y configuración de QMK	14
6. Desarrollo del software	15
6.1. Configuración inicial	15
7. Anexo I: Instalación de MicroPython	18
7.1. Preparar el compilador	18
7.2. Compilar para Linux (Opcional)	18
7.3. Compilar para RP2040	18
8. Referencias	22

1.1. Motivación

Hoy en día, es mucha la gente que se pasa buena parte del día frente a un ordenador, tanto por trabajo como en su tiempo libre. Esto, por supuesto, puede suponer problemas para la salud si no se toman las precauciones necesarias. Por ejemplo, problemas de vista por pasar excesivas horas mirando un monitor, aunque en este frente ya hay una buena cantidad de divulgación e investigación.

Sin embargo, el periférico que más usamos es el teclado y, sin embargo, su *problemático* diseño apenas ha cambiado desde que existen los ordenadores y puede resultar en diversas lesiones y enfermedades en las muñecas.

1.2. Objetivo

El principal fin de este trabajo va a ser el diseño de un teclado que se adapte mejor a la anatomía humana y que, a su vez, incorpore mejoras que lo hagan accesible a gente con diversas discapacidades e integración con la domótica de casa **Home Assistant** [1]:

- Teclas con letras grandes y alto contraste
- Un joystick que permite escribir
- Software propio para
 - Ajustar la configuración del firmware
 - Permitir comunicaciones con otro software/hardware

2.1. Requisitos previos

En la siguiente sección, se asume que ya tenemos *git* [2], *Python* [3] y *Visual Studio Code* [4] instalados. En el editor podemos ir instalando distintas extensiones para autocompletado de sintaxis en lenguajes que no estén soportados de base y herramientas de ayuda. Estos complementos no son **necesarios** y su instalación es tan trivial como ir al apartado de *extensiones*, por lo que no comentaré nada sobre ellos.

2.2. Configuración para trabajar en remoto

El desarrollo de este trabajo ha sido realizado en una máquina Arch Linux, controlada por SSH desde Windows, por lo que en el informe se verán comandos, imágenes y archivos que mezclan ambos sistemas operativos. Salvo que se especifique lo contrario, cualquier adjunto pertenece a Linux.

En otras distribuciones de Linux, habría que usar otros gestores de paquetes, como `> apt-get` o `> zypper` y algun programa podría estar empaquetado con otro nombre.

2.2.1. SSH

Para poder trabajar cómodamente en la terminal, generamos una clave SSH en Windows con `> ssh-keygen` y la salida del comando se guarda en `C:/Users/<usuario>/ssh/id_<encriptado>.pub`

El contenido de este archivo lo copiamos en Linux en `~/.ssh/authorized_keys`, gracias a esto podremos conectarnos en remoto a Linux sin necesidad de introducir las credenciales, puesto que hemos añadido la identidad de la máquina Windows como un dispositivo de confianza.

2.2.2. Montar disco en red

Con el fin de acceder de forma cómoda y rápida a los archivos Linux, en vez de estar usando SCP o SSH, montaremos una ubicación de red en Windows. Instalamos con `> sudo pacman -S samba` y activamos los servicios `> sudo systemctl enable smb` y `> sudo systemctl enable nmb`. Tras crear el usuario e introducir en Windows las credenciales, tenemos acceso a todo el `$HOME` de nuestro Linux:

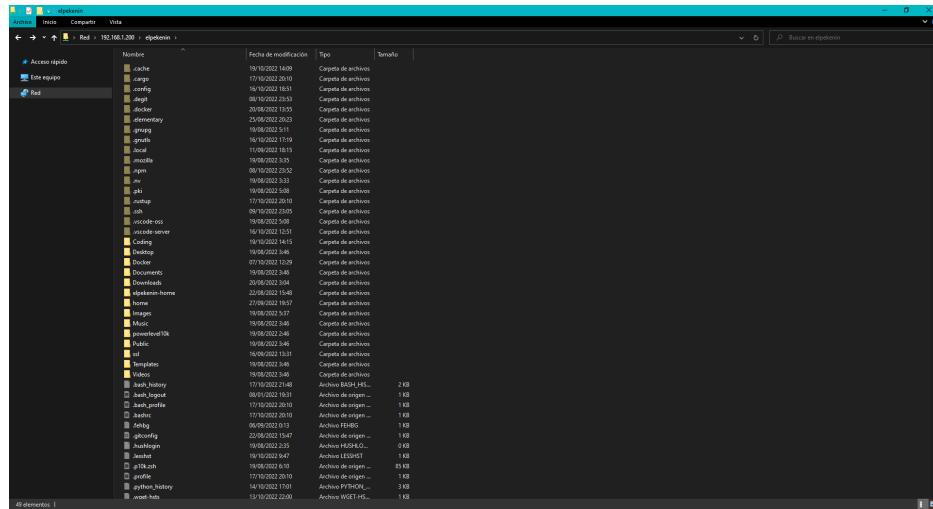


Figura 1: Carpeta montada en Windows

2.2.3. git

Al intentar usar la carpeta recién montada, surge un problema puesto que la instalación de git no confía en el repositorio, para arreglar esto debemos ejecutar

```
> git config --global --add safe.directory '%(prefix)//192.168.1.200/elpekenin/Coding/access_kb'
```

2.2.4. Servidor X

Para poder desarrollar el programa en Linux y testearlo desde Windows, debemos instalar un servidor que nos proporcione compatibilidad con la interfaz gráfica de Linux, he usado **VcXsrv** [5]. A partir de ahora cuando nos conectemos a la máquina de desarrollo, añadimos la opción **-X** al comando SSH, para habilitar el redireccionamiento de los gráficos(por defecto está activa, pero prefiero añadirla por si acaso)

2.2.5. L^AT_EX

Este informe está escrito en L^AT_EX, para usarlo instalamos el software necesario con `> sudo pacman -S texlive-most`

Y para ver el PDF resultante, instalamos un visor minimalista `> sudo pacman -S mupdf`.

Como el compilado de L^AT_EX incluye varios pasos, añadiremos una función a la configuración de shell (en mi caso, ZSH)

```
</> ~/.zshrc

pdf () {
    echo 'Cleaning previous compilation files ...'
    rm /tmp/main*

    echo 'Compiling ...'
    cd "${_MY_PROJECT_PATH}latex" && pdflatex -output-directory=/tmp main.tex > /dev/null 2>&1

    echo 'Adding references...'
    biber /tmp/main.bcf > /dev/null 2>&1
```



```
echo 'Recompiling ...'
cd "${_MY_PROJECT_PATH}latex" && pdflatex -output-directory=/tmp main.tex > /dev/null 2>&1

echo 'Copying file to working directory ...'
cp /tmp/main.pdf report.pdf

if pgrep -x "mupdf" > /dev/null
then
    echo 'Killing mupdf ...'
    pkill mupdf
fi

echo 'Opening report ...'
(
    mupdf report.pdf > /dev/null 2>&1 &
)
}
```

Este comando se ejecuta haciendo simplemente `> pdf`. Funciona de forma silenciosa, redirigiendo la salida de los comandos para que no salgan por pantalla; y al terminar abre el archivo producido.

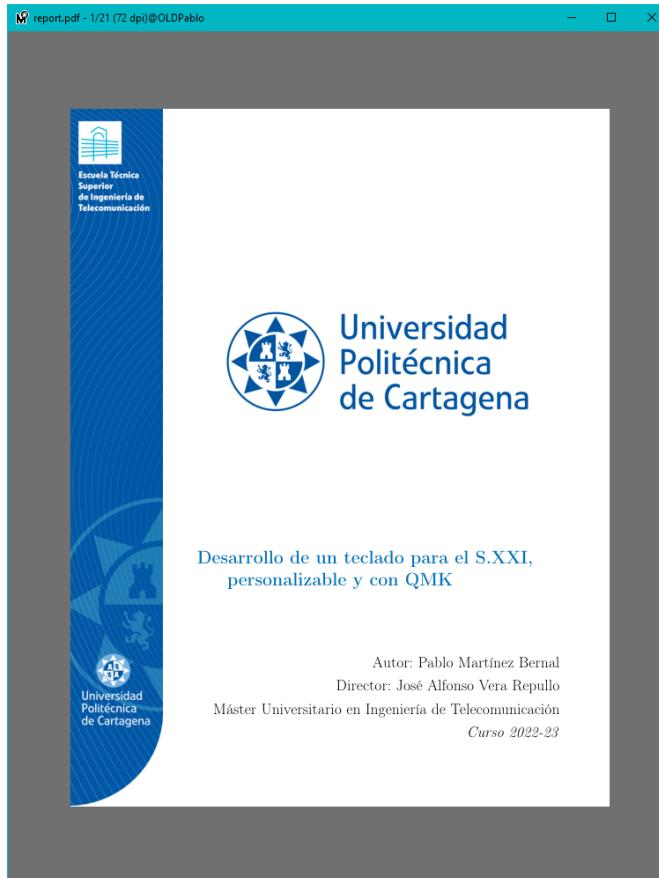


Figura 2: Viendo el PDF en Windows

SECCIÓN 3

ESTADO DEL ARTE

3.1. Diseño



(a) Teclado IBM Model M (1984)



(b) Teclado Logitech MX Mechanical (2022)

Figura 3: Comparativa teclado antiguo y actual



Figura 4: Teclado en Android

Como podemos ver, los teclados no han variado en los últimos 40 años, ni siquiera para adaptarse a las pequeñas pantallas táctiles de los móviles.

3.1.1. Diseño anticuado

Forma del teclado

Quizás nunca nos lo hayamos preguntado puesto que tenemos muy interiorizada la forma de los teclados, pero si lo pensamos un poco, es peculiar la posición relativa de las teclas. Cada fila tiene un pequeño desfase con las demás en vez de estar alineadas. Esto es un legado de sus antecesoras, las máquinas de



escribir, donde por limitaciones mecánicas esto tenía que ser así y evitar choques entre las piezas móviles de cada tecla.



Figura 5: Detalle de las letras en una máquina de escribir

Este posicionamiento relativo de las teclas supone un problema a la hora de escribir, ya que la forma óptima de hacerlo sería la siguiente:



Figura 6: “Mapa” de mecanografía

Sin embargo, para escribir así, las muñecas terminan en posiciones un poco forzadas, y los dedos hacen movimientos incómodos. Para arreglar esto surgieron los teclados ortolineales, donde todas las filas están alineadas y los dedos se mueven en una línea recta. Estos teclados suelen tener todas las teclas del mismo tamaño, optimizando así la cantidad de teclas que podemos tener ocupando el mismo espacio (donde antes había una barra espaciadora pueden entrar varias teclas). Como se puede ver en la siguiente imagen, normalmente también prescinden del teclado numérico para reducir el tamaño, este modelo se conoce como “75 %” ya que tiene 75 teclas mientras que los teclados comunes (“100 %”) tienen 104/105 teclas. Otras variantes comunes son “40 %”, “60 %”, “65 %”



Figura 7: Teclado ortolineal *RGB75*

Ubicación de las letras

Otro legado que nos dejaron las máquinas de escribir es la distribución QWERTY, que probablemente sea la única distribución que hemos visto a lo largo de nuestra vida. El problema con esta disposición es que, si bien distribuye las letras de forma que se usan las dos manos por igual, se diseñó en la década de

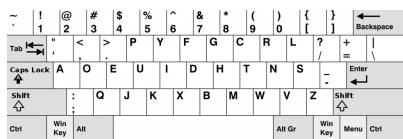


1860, por lo que uno de sus objetivos era el de reducir los atascos en las máquinas de escribir separando las teclas más usadas de la parte central.

En contra, ahora que gracias a la electrónica no tenemos estas limitaciones, se han diseñado distribuciones que minimizan la distancia media que se debe recorrer al escribir, por lo que una vez acostumbrados a ellas se puede escribir más rápido y reduciendo la fatiga en los dedos. Las dos más extendidas son DVORAK y COLEMAK.



Figura 8: Distribución QWERTY



(a) Dvorak



(b) Colemak

Figura 9: Distribuciones alternativas

3.1.2. Algunos avances

Gracias a la apasionada y extensa comunidad de aficionados a los teclados mecánicos, en los últimos años han aparecido multitud de diseños que incorporan diferentes cambios respecto al paradigma actual, algunos de estos conceptos son:

Split

Como hemos comentado ya, uno de los problemas más comunes en gente que usa mucho los teclados es la aparición de dolencias en las muñecas, a fin de que estas se posicen de una forma más natural y cómoda, se opta por partir el teclado en dos mitades.



Figura 10: Teclado *Quefreny*

Pulgares

Muchos teclados dotan de una mayor utilidad a los pulgares, que normalmente solo utilizamos para la barra espaciadora, añadiendo unas cuantas teclas en lo que comunmente se conoce como *thumb cluster*.



Figura 11: Teclado *Ergodox*

Reposamuñecas

Otros diseñadores aprovechan la oportunidad para integrar este accesorio de una forma más eficaz que la típica “rampa” de plástico que estamos acostumbrados a ver.



Figura 12: Teclado *Moonlander*



Tenting

Esta conocida técnica solo se puede aplicar en teclados *split* y consiste en levantar la parte central del teclado, de forma que la muñeca está en una posición más natural en vez de estar paralela al plano que genera la mesa. Lo mejor de esta mejora es que se puede añadir a cualquier teclado que no la incorpore en su diseño añadiendo algún objeto para levantarla.



Figura 13: Teclado *Dygma Raise*

Ergonomía

El mayor ejemplo de esta filosofía de diseño es el *Dactyl Manuform*, un teclado que debido a su particular forma ni siquiera puede funcionar con una PCB (placa de circuito impreso) y tiene que soldarse a mano la unión entre todos sus componentes. El beneficio de su diseño es que tiene en cuenta la forma de las manos, por lo que las teclas se encuentran posicionadas acorde al movimiento de los dedos. Además, hay usuarios que optan por modificar el diseño e integrarle una *trackball* para poder controlar el cursor sin tener que mover la mano entre el teclado y el ratón.



Figura 14: Teclado *Dactyl Manuform* con trackball

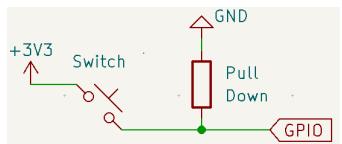
3.2. Firmware

3.3. Hardware

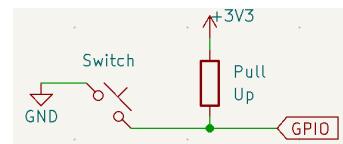
4.1. Cableado de las teclas

4.1.1. Conexión directa

La opción más sencilla que se nos puede ocurrir para conectar diversos interruptores a nuestro microcontrolador y hacer un teclado es usar cada uno de los pines de entrada/salida(GPIO) para registrar el estado de cada pulsador. Para hacer esto tenemos 2 opciones:



(a) Pulsador con resistencia Pull-Down



(b) Pulsador con resistencia Pull-Up

Figura 15: Cableado directo

Si hacemos esto, sin embargo, tendremos un problema pronto, y es que necesitaremos un chip con muchos pines de entrada salida, o hacer un teclado con pocas teclas porque la cantidad de GPIOs no es ilimitada.

4.1.2. Matriz

Para solventar este problema, podemos cablear los botones mediante una matriz, necesitando un pin para cada fila y columna de teclas. En este caso, usaríamos una dimensión como salida y otra como entrada. Haciendo un bucle que aplique voltaje en cada una de las filas y comprobando si las diversas columnas tienen entrada (tecla pulsada cerrando el circuito) o no. *La iteración se podría hacer en el sentido contrario*

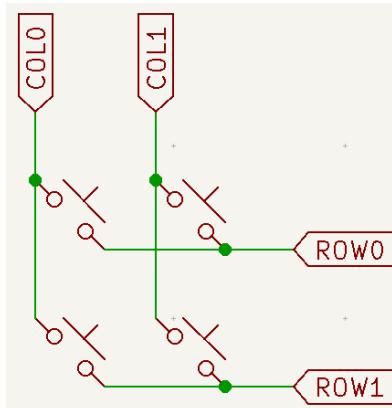


Figura 16: Cableado en matriz



Sin embargo, este diseño también tiene sus problemas. El más notorio es el conocido como “efecto *ghosting*” en el que podemos detectar como pulsada una tecla que no lo está.

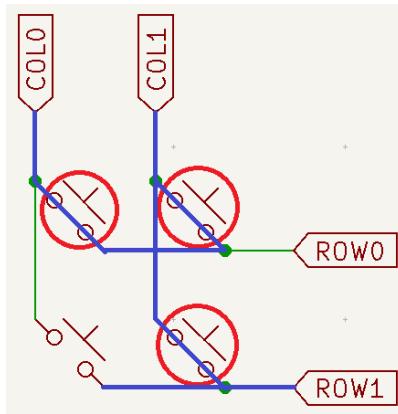


Figura 17: Ghosting en una matriz

En este ejemplo, la tecla **(1, 1)** se detecta como pulsada de manera correcta. Sin embargo, al pulsar la tecla **(0, 1)** estamos cerrando el circuito y generando que el nodo de la fila 0 también tenga voltaje, por lo que, al estar pulsada la tecla **(0, 0)** estamos haciendo que en la columna 1 llegue una entrada que será detectada como que la tecla **(1, 0)** ha sido pulsada puesto que estamos en la iteración de la fila 0.

Este problema se solventa de forma sencilla, añadiendo unos diodos que bloquean esta retroalimentación, detectando correctamente **(1, 1)** sin la pulsación falsa de **(1, 0)** del caso anterior.

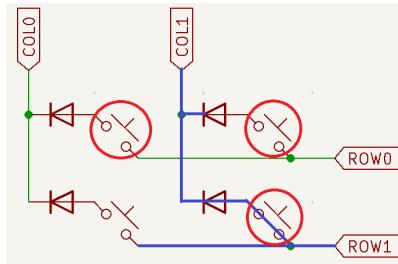


Figura 18: Matriz anti-ghosting

5.1. Instalación y configuración de QMK

Para instalar su CLI ejecutamos `> pip install qmk` ya que se trata de una librería escrita en Python y disponible en el repositorio de paquetes de Python(pip).

Tras esto, descargamos el repositorio de QMK con `> qmk setup`, a este comando le podemos pasar como parámetro nuestro fork del repositorio (en mi caso `> qmk setup elpekenin/qmk_firmware`), para poder usar git, ya que no tendremos permisos en el repositorio oficial. Este comando también nos instalará los compiladores necesarios y comprobará las udev de nuestro sistema Linux, para que podamos trabajar con los dispositivos sin problema, en caso de que no estén bien configuradas podemos usar copiar el archivo `50-qmk.rules` en `/etc/udev/rules.d/`. Finalmente podemos ejecutar `> qmk doctor` para comprobar el estado de QMK. Si estamos en Linux, es posible que tengamos problemas con `udev`, las reglas que controlan los permisos sobre dispositivos. Para solventar esto copiamos el archivo `50-qmk.rules` en `/etc/udev/rules.d/` Para poder hacer debug con `> qmk console` he necesitado añadir una línea a dicho archivo:

```
KERNEL=="hidraw", SUBSYSTEM=="hidraw", MODE=="0666", TAG+="uaccess", TAG+="udev-acl" Options
```

adicionalmente, podemos instalar LVGL funciones gráficas más complejas en la pantalla LCD, en vez de usar el driver de QMK. Esto lo haremos con

```
> git submodule add -b release/v8.2 https://github.com/lvgl/lvgl.git lib/lvgl
```

(desde el directorio base de QMK). Seguidamente, usamos el código de **jpe230 [6]** para usar esta librería.

Errores de compilación

Es posible que al intentar compilar obtengamos un error parecido a `error: array subscript 0 is outside array bounds of 'uint16_t[0]' [-Werror=array-bounds]`, este error se debe a un cambio en **gcc 12**. Seguramente estará solucionado en un par de meses con una actualización en QMK, pero de no ser así podemos revertir a **gcc 11** para que el mismo código se pueda compilar.

Una vez configurado QMK, creamos los archivos básicos para el firmware de nuestro teclado haciendo `> qmk new-keyboard` e introduciendo los datos necesarios, sin embargo esto crea una carpeta directamente en la ruta `qmk_firmware/keyboards`, en mi caso he creado una carpeta nueva bajo este directorio con mi nick (elpekenin) como nombre, y he movido la carpeta del teclado ahí dentro, de forma que si en un futuro diseño otro teclado se guarde en esta misma carpeta.

Para acelerar el proceso de compilado podemos guardar nuestra configuración (teclado, keymap y número de hilos que se usan):

```
> qmk config user.keyboard=elpekenin/access
> qmk config user.keymap=default
> qmk config compile.parallel=8
```

Para poder intercambiar información con el teclado de forma que podamos configurarlo o enviarle información en vez de simplemente escuchar las teclas que se han pulsado, vamos a desarrollar un programa que se ejecute en el ordenador.

Vamos a usar **Tauri** [7] ya que permite usar el mismo código en multitud de sistemas operativos, esto se consigue gracias a que funciona internamente con un servidor HTML, por lo que se puede ejecutar en diversas plataformas.

6.1. Configuración inicial

Para instalar astro

```
# Dependencias
> sudo pacman -Syu
> sudo pacman -S --needed \
    webkit2gtk \
    base-devel \
    curl \
    wget \
    openssl \
    appmenu-gtk-module \
    gtk3 \
    libappindicator-gtk3 \
    librsvg \
    libvips
> curl -proto '=https' -tlsv1.2 https://sh.rustup.rs -sSf | sh
> cargo install tauri-cli

# Crear proyecto
> npm create tauri-app
```

Instalamos **Node.js** [8] con `> sudo pacman -S nodejs` para poder usar **Astro** [9], que es el framework con el que haremos el frontend (interfaz gráfica) de la aplicación y creamos el proyecto con `> npm create astro@latest`. Tras esto, modificamos el archivo de configuración de Tauri para que execute el servidor de Astro:

```
</> software/src-tauri/tauri.conf.json

"build": {
    "beforeDevCommand": "npm run dev",
    "beforeBuildCommand": "npm run build",
```



```
"devPath": "http://localhost:8000",  
"distDir": "../dist",
```

También tenemos que editar la configuración de Astro para que se ejecute en el mismo puerto que hemos configurado en Tauri, y de paso añadimos la opción **host** para que el servidor Astro sea visible desde otros dispositivos de la red local y poder hacer debugging de HTML/JS/CSS sin pasar por Tauri.

```
</> software/frontend/astro.config.mjs
```

```
+ server: {  
+   host: true,  
+   port: 8000  
+ }
```

En este momento tenemos la siguiente estructura de archivos:

```
> tree -L 2  
.---- astro.config.mjs  
.---- package.json  
.---- package-lock.json  
.---- src  
|---- components  
|---- env.d.ts  
|---- layouts  
|---- pages  
|---- src-tauri  
|---- build.rs  
|---- Cargo.lock  
|---- Cargo.toml  
|---- icons  
|---- src  
|---- target  
|---- tauri.conf.json  
---- tsconfig.json
```

Ejecutando `> cargo tauri dev` podemos lanzar nuestro programa.

He tenido que desactivar la opción wgl(libreria de Windows para OpenGL) en VcXsrv para que funcione

En este punto añadimos **hidapi** [10] para poder usar el protocolo **HID** [11] desde Rust, con el que nos comunicaremos con el teclado.

```
</> src-tauri/Cargo.toml
```

```
[dependencies]  
+ hidapi = "1.4.2"
```



Para poder ejecutar código Rust desde JavaScript instalamos el módulo necesario

```
> npm i @tauri-apps/api
```

y la integración de *Svelte* [12] con Astro y lo habilitamos con

```
</> /astro.config.mjs

+ import svelte from "@astrojs/svelte";
  ...
+   integrations: [svelte()]
```

Definimos un nuevo componente

```
</> src/components/Call.svelte

<script lang="ts">
  import { invoke } from '@tauri-apps/api';

  export let command = "greet";
  export let args = {"name": "peke"};

  try { invoke(command, args); }
  catch (e) { console.error(e); }
</script>
```

Con este nuevo componente podremos hacer `<Call command="function" args={{key: value}}>` para ejecutar cualquiera de nuestras funciones de Rust pasándole los parámetros necesarios.

SECCIÓN 7

ANEXO I: INSTALACIÓN DE MICROPYTHON

Durante el desarrollo del proyecto, he probado MicroPython, una implementación en C del intérprete de Python que se enfoca a su uso en microcontroladores. Finalmente he descartado usarlo debido a su menor rendimiento y la falta de muchas opciones que vienen hechas en QMK. Sin embargo creo que puede ser una buena alternativa para prácticas de electrónica en la universidad, reemplazando a Arduino, puesto que Python es mucho más amigable que C o C++.

7.1. Preparar el compilador

Tras clonar el source de MicroPython `> git clone https://github.com/miropython/micropython`, hacemos `> make -C mpy-cross` para compilar el compilador cruzado de MicroPython que nos permitirá convertir el código fuente para ser ejecutado en diferentes arquitecturas.

7.2. Compilar para Linux (Opcional)

Si queremos usar MicroPython en nuestro ordenador para hacer pruebas, en vez de CPython(que es la versión más común), usaremos el compilador que acabamos de construir para compilar el código fuente del intérprete y usarlo en nuestra máquina

```
> cd ports/unix  
> make submodules  
> make
```

Ya podemos ejecutar MicroPython

```
> cd build-standard  
> ./micropython  
MicroPython 13dceaa4e on 2022-08-24; linux [GCC 12.2.0] version  
Use Ctrl-D to exit, Ctrl-E for paste mode
```

7.3. Compilar para RP2040

Primero instalamos un compilador necesario para la arquitectura del procesador, en mi caso (Arch Linux), el comando es `> sudo pacman -S arm-none-eabi-gcc` y después añadimos la configuración necesaria para reportar y usar un endpoint HID, siguiendo (y adaptando) el código de **noobee [13]**. Definimos en C el módulo `usb_hid` y con `MP_REGISTER_MODULE` lo añadimos al firmware



```
</> ports/rp2/modusb_hid.c

/*
 * This file is part of the MicroPython project, http://micropython.org/
 *
 * The MIT License (MIT)
 *
 * Copyright (c) 2020-2021 Damien P. George
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the "Software"),
 * to deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or
 * sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
 *
 * ,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
 * IN
 * THE SOFTWARE.
 */

#include "py/runtime.h"
#include "tusb_config.h"

extern bool usbd_tud_hid_report(uint8_t report_id, void const* report,
                                uint8_t len);

STATIC mp_obj_t usb_hid_report(mp_obj_t report_id_obj, mp_obj_t report_obj)
{
    uint8_t report_id = mp_obj_get_int(report_id_obj);
    mp_buffer_info_t report;
    mp_get_buffer_raise(report_obj, &report, MP_BUFFER_READ);

    if (report_id < REPORT_ID_KEYBOARD || report_id > REPORT_ID_GAMEPAD) {
        mp_raise_msg_varg(&mp_type_ValueError, MP_ERROR_TEXT("report_id
```



```
invalid (%d)", report_id);
}

usbd_tud_hid_report(report_id, report.buf, report.len);
return mp_const_none;
}
MP_DEFINE_CONST_FUN_OBJ_2(usb_hid_report_obj, usb_hid_report);

STATIC const mp_rom_map_elem_t usb_hid_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_usb_hid)
    },

    { MP_ROM_QSTR(MP_QSTR_KEYBOARD), MP_ROM_INT(REPORT_ID_KEYBOARD)
    },
    { MP_ROM_QSTR(MP_QSTR_MOUSE), MP_ROM_INT(REPORT_ID_MOUSE)
    },
    { MP_ROM_QSTR(MP_QSTR_MOUSE_ABS), MP_ROM_INT(
REPORT_ID_MOUSE_ABS) },
    { MP_ROM_QSTR(MP_QSTR_CONSUMER_CONTROL), MP_ROM_INT(
REPORT_ID_CONSUMER_CONTROL) },
    { MP_ROM_QSTR(MP_QSTR_GAMEPAD), MP_ROM_INT(REPORT_ID_GAMEPAD)
    },
    { MP_ROM_QSTR(MP_QSTR_report), MP_ROM_PTR(&
usb_hid_report_obj) },
};

STATIC MP_DEFINE_CONST_DICT(usb_hid_module_globals,
    usb_hid_module_globals_table);

const mp_obj_module_t mp_module_usb_hid = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&usb_hid_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_usb_hid, mp_module_usb_hid);
```

Editamos este archivo para que el compilador añada `modusb_hid.c`

```
</> ports/rp2/CMakeLists.txt

set(MICROPY_SOURCE_PORT
+ modusb_hid.c
fatfs_port.c

set(MICROPY_SOURCE_QSTR
+ ${PROJECT_SOURCE_DIR}/modusb_hid.c
${MICROPY_SOURCE_PY})
```

Para poder compilar la versión de RP2040, también he necesitado instalar otro paquete

```
> sudo pacman -S arm-none-eabi-newlib
```



Por último, compilamos con

```
> cd ports/rp2  
> make submodules  
> make clean  
> make
```

SECCIÓN 8

REFERENCIAS

1. *Home Assistant*, <https://www.home-assistant.io/>.
2. *git*, <https://git-scm.com/>.
3. *Python*, <https://www.python.org/downloads/>.
4. *Visual Studio Code*, <https://code.visualstudio.com/>.
5. *VcXsrv*, <https://sourceforge.net/projects/vcxsrv/>.
6. jpe230, *Añadir LVGL en QMK*, https://github.com/Jpe230/qmk_firmware/commits/develop-lvgl.
7. *Tauri*, <https://tauri.app/>.
8. *Node.js*, <https://nodejs.org/en/>.
9. *Astro*, <https://astro.build/>.
10. *hidapi*, <https://docs.rs/hidapi/latest/hidapi/>.
11. *HID*, <https://es.wikipedia.org/wiki/HID>.
12. *Svelte*, <https://svelte.dev/>.
13. noobee, *Añadir HID a RP2040 en MicroPython*, <https://github.com/noobee/micropython/tree/usb-hid>.