



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. Ergonomía	9
3.2. Hardware	12
3.2.1. Cableado de las teclas	12
3.2.2. Pantallas	14
3.2.3. Sensor táctil	14
3.3. Firmware	15
3.3.1. Funcionalidad	15
3.3.2. Escaneo de teclas	15
3.3.3. Pantallas	15
3.3.4. Pantalla táctil	15
4. Desarrollo	16
4.1. Hardware	16
4.1.1. Objetivos	16
4.1.2. Distribución	16
4.2. Firmware	16
4.2.1. Instalación y configuración de QMK	16
4.2.2. Driver para pantalla ili9486	17
4.2.3. Dibujar por USB	18
4.2.4. Mejora algoritmo elipses	18
4.2.5. Driver para sensor XPT2046	18
4.2.6. Generar imágenes	18
4.2.7. Control con una mano	19
4.2.8. Añadir nuestro código	19
4.2.9. Comunicación con el software de PC	19



4.3.	Software	20
4.3.1.	Instalación	20
4.3.2.	Desarrollo	21
5.	Anexo I: Instalación de MicroPython	26
5.1.	Preparar el compilador	26
5.2.	Compilar para Linux (Opcional)	26
5.3.	Compilar para RP2040	26
6.	Referencias	29

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, a modo de prueba de concepto se implementará un modo (bastante básico) que permita controlar el teclado haciendo uso de una mano/dedo para que sea más accesible a gente con alguna discapacidad.

También vamos a desarrollar un software propio que permita la fácil integración con la domótica[1] de casa (y cualquier otro servicio). Esto es especialmente interesante con vistas a futuro, ya que con el auge del IoT cada vez tenemos más dispositivos por casa con los que sería conveniente tener una manera cómoda de comunicarnos. Utilidades del programa:

- Ajustar la configuración del firmware, como cambiar la acción de las teclas sin tener que compilar y flashear
- Permitir que el teclado se comunique con el ordenador, de forma que podamos
 - Controlar la domótica (y otras acciones) mediante *triggers* que hagamos en el teclado. Por ejemplo, pulsar una combinación de teclas para encender la luz
 - Enviar información desde el ordenador. Por ejemplo, para mostrar en las pantallas del teclado la fecha o temperatura actual.

SECCIÓN 2

PREÁMBULO

2.1. Requisitos previos

En la siguiente sección, se asume que ya tenemos algunos programas instalados (git, Python, Visual Studio Code...). En este 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.

Parte del desarrollo de este trabajo ha sido realizado en una máquina con ArchLinux, controlada por SSH desde Windows. Aunque posteriormente he migrado a WSL[2] por lo que en el informe se verán comandos, imágenes y archivos que mezclan ambos sistemas operativos. Para la instalación de cualquier programa en el correspondiente SO podemos seguir su documentación oficial, por lo que solo comentaré cosas que no queden del todo claras.

2.2. Configuración para trabajar en remoto

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 la ubicación `~/.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:

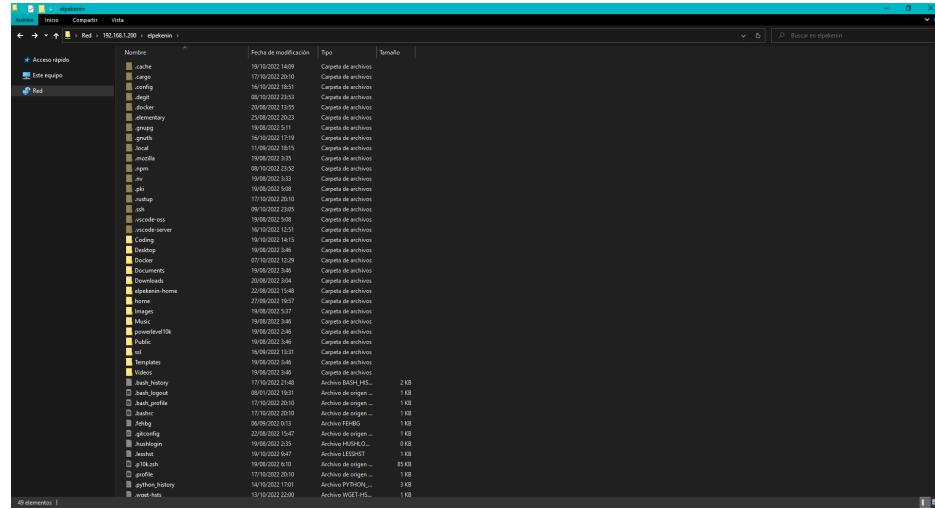


Figura 1: Carpeta montada en Windows

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 ejecutamos un comando, en mi caso fue:

```
> 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[3]. 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
```

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[4] a la configuración de shell. 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.

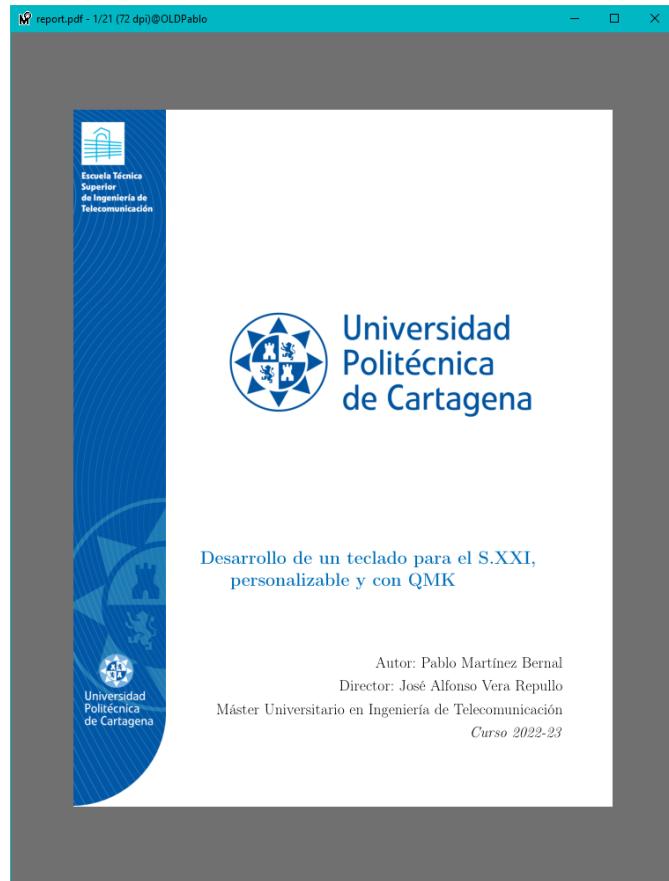


Figura 2: Viendo el PDF en Windows

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

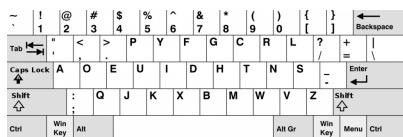


1860, por lo que uno de sus objetivos era el de reducir los atascos en las máquinas de escribir separando las teclas más usadas de la parte central.

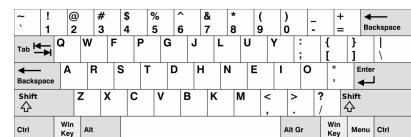
En contra, ahora que gracias a la electrónica no tenemos estas limitaciones, se han diseñado distribuciones que minimizan la distancia media que se debe recorrer al escribir, por lo que una vez acostumbrados a ellas se puede escribir más rápido y reduciendo la fatiga en los dedos. Las dos más extendidas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



Figura 8: Distribución QWERTY



(a) Dvorak



(b) Colemak

Figura 9: Distribuciones alternativas

3.1.2. Ergonomía

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.

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, lo que se conoce como teclados *split*.



Figura 10: Teclado *Quefrency*

Otra técnica, mayormente usada en teclados *split*, consiste en levantar la parte central del teclado, de forma que la mano quede en una posición más natural en vez de estar paralela al plano que forma la



mesa. Lo mejor de esta mejora es que se puede añadir a cualquier teclado añadiendo algún objeto para levantararlo.



Figura 11: Teclado *Dymga Raise*

Otros diseñadores integran reposamuñecas de una forma más eficaz y cómoda que la típica “rampa” de plástico que estamos acostumbrados a ver.



Figura 12: Teclado *Moonlander*

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 comúnmente se conoce como *thumb cluster*.



Figura 13: Teclado *Ergodox*

El mayor ejemplo de estas ideas es el *Dactyl Manuform*, un teclado que debido a su particular forma ni siquiera puede funcionar con una PCB 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

En esta web^[5] se pueden ver varios estudios sobre la relación entre el diseño del teclado y sus efectos en la salud

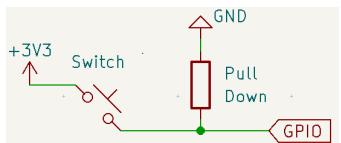


3.2. Hardware

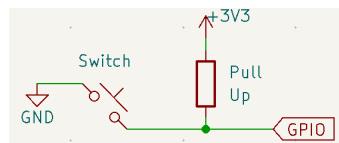
3.2.1. Cableado de las teclas

Conexión directa

La opción más sencilla que se nos puede ocurrir para conectar diversos interruptores a nuestro microcontrolador es soldarlos directamente a los pines de entrada/salida(GPIO). 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 porque necesitaremos un chip con muchos pines de entrada/salida, o hacer un teclado con pocas teclas porque la cantidad de GPIOs es reducida.

Matriz

Para solventar este problema, podemos cablear los botones mediante una matriz, usando un pin para cada fila y columna de teclas. Usamos una dimensión como salida y otra como entrada, haciendo un bucle que aplique voltaje en cada una de las filas y compruebe si las columnas reciben una entrada (tecla pulsada cerrando el circuito). *Nota: También se podría iterar en la otra dimensión*

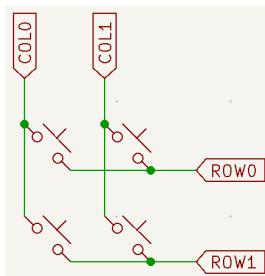


Figura 16: Cableado en matriz

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.

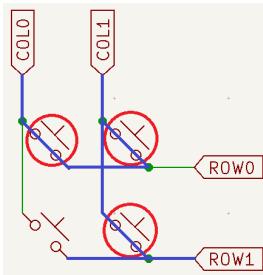


Figura 17: Ghosting en una matriz

En este ejemplo, la tecla **(1, 1)** se detecta como pulsada de manera correcta pero, al pulsar también las teclas **(0, 1)** y **(0, 0)**, estamos cerrando el circuito y generando que en la columna 1 llegue voltaje a la entrada, que será interpretado 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 bloquen esta retroalimentación permitiendo detectar **(1, 1)** pero sin la pulsación falsa de **(1, 0)** del caso anterior.

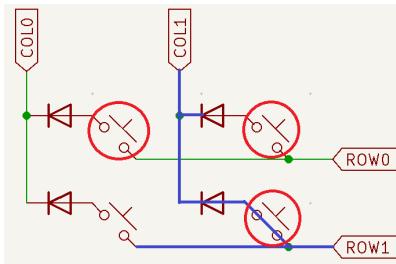


Figura 18: Matriz anti-ghosting

Aunque no es muy grave, para matrices muy desiguales, por ejemplo 5x20 teclas, no estamos usando eficazmente los pines, ya que para esas 100 teclas estamos empleando 25 pines mientras que una configuración 10x10(20 pines) sería suficiente.

En este caso, podríamos hacer una distribución de teclas en forma rectangular, pero luego cablearlas como dicha matriz cuadrada, sin embargo el diseño sería bastante más confuso.

Por último hay que tener en cuenta que, aunque el uso de los pines sea más óptimo, seguimos necesitando una cantidad de pines cada vez mayor conforme queramos añadir más teclas, aunque esto no debería ser un factor limitante en la mayoría de casos ya que este límite seguiría permitiendo una cantidad bastante elevada de teclas.

Lectura en serie

La opción que vamos a usar, inspirada en el *ghoul*[6], consiste en el uso de registros de desplazamiento conectados en una *daisy chain*, de esta forma vamos a emplear un único pin para leer todas las teclas en una señal serie, y otros pocos pines (unos 3 o 4) para controlar estos chips mediante SPI[7] o I2C[8]. De esta forma, podemos escanear potencialmente cualquier cantidad de teclas sin aumentar el números de pines necesarios, simplemente añadimos más registros a la cadena (aunque el escaneo se iría haciendo más y más lento)



3.2.2. Pantallas

En los últimos años es cada vez más común ver teclados que incorporan pequeñas pantallas, sin embargo, no son muy útiles ya que en la gran mayoría de casos se trata de SH1106 o SSD1306, que son dispositivos OLED de 2 colores y con una resolución bastante reducida, de 128x32 o 128x64 píxeles, en torno a la pulgada de diagonal. En nuestro teclado vamos a usar pantallas más potentes para poder mostrar información útil en vez de estar limitados a pequeños dibujos en estas pantallas “convencionales”

3.2.3. Sensor táctil

También es relativamente común encontrar diseños que incluyen diferentes sensores (por ejemplo joysticks analógicos o PMW3360) para mover el cursor por la pantalla del ordenador sin tener que mover la mano hasta el ratón. Tal como podemos ver en el dactyl (3.1.2)

En nuestro caso, puesto que vamos a añadir una pantalla, podemos aprovechar y usar una táctil, de forma que nos sirva para mover el cursor pero también para tener una pequeña interfaz de usuario en el teclado.



3.3. Firmware

3.3.1. Funcionalidad

La mejor parte de usar una librería tan extendida como lo es QMK es que tenemos muchas facilidades a la hora de escribir el código ya que buena parte del trabajo está hecho ya. Esto incluye

- Gestión sobre USB para reportar distintos endpoints (teclado, ratón, multimedia)
- Drivers para diversos periféricos tales como las pantallas ya comentadas, piezoelectricos para tener feedback sonoro o solenoides para vibración al pulsar las teclas, por nombrar algunos...
- Abstracciones para poder usar la misma API en diversos microcontroladores que internamente utilizan código y hardware muy diferente
- Documentación extensa y detallada
- Servidor oficial en Discord donde podemos encontrar ayuda

3.3.2. Escaneo de teclas

Como se ha comentado en el apartado anterior, vamos a usar registros de desplazamiento, sin embargo QMK tiene soporte para cableado directo, en matrices y usando algunos otros circuitos, como *IO expanders*. Sin embargo este no es el caso para los registros por lo que tendremos que escribir un poco de código para leer su información.

3.3.3. Pantallas

QMK tiene una API estandarizada (Quantum Painter[9]) para primitivas de dibujo en interfaces gráficas. Aún es “joven” y no soporta demasiadas pantallas, pero hace gran parte del trabajo por nosotros.

3.3.4. Pantalla táctil

QMK también tiene una capa de abstracción[10] para diversos sensores que permiten mover el cursor, sin embargo todos se basan en medir desplazamientos y no una coordenada como es el caso de la pantalla táctil, por tanto en este caso nos vemos obligados a diseñar una arquitectura de software nueva, para poder trabajar con este otro tipo de datos de entrada.

SECCIÓN 4

DESARROLLO

4.1. Hardware

4.1.1. Objetivos

Las metas principales a la hora de diseñar el teclado han sido el usar la menor cantidad de pines posible para las tareas “básicas” y el exponer todos los pines restantes así como varias tomas de alimentación. De esta forma, el teclado puede servir como placa de pruebas donde desarrollar drivers para otro hardware y además es modular, ya que nos permite añadir más periféricos (por ejemplo, para sonido) en el futuro sin necesidad de tener que fabricar una nueva PCB.

4.1.2. Distribución

He optado por una disposición ortolineal, split y (por ahora) QWERTY, reduciendo un par de columnas respecto al tamaño habitual ya que algunas de esas teclas raramente se usan, y al tener una forma simétrica es más sencillo de diseñar.



Figura 19: Diseño aproximado del teclado

4.2. Firmware

4.2.1. Instalación y configuración de QMK

Para usar QMK[11] instalamos su CLI ejecutando `> pip install qmk` ya que se trata de una librería escrita en Python y está disponible en el repositorio de paquetes de Python(pip).



Tras esto, descargamos el código fuente 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.

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"
```

Opcionalmente, podríamos instalar LVGL[12] funciones gráficas más complejas en la pantalla LCD, en vez de usar el driver de QMK. Esto se hace 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 jpe[13] para usar esta librería.

De momento no he implementado esta librería puesto que añadiría bastante complejidad a **4.2.3 Dibujar por USB**

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. Pero también podemos arreglarlo manualmente con:

```
> sudo pacman --needed -U https://archive.archlinux.org/packages/a/arm-none-eabi-gcc/arm-none-eabi-gcc-11.3.0-1-x86_64.pkg.tar.zst
```

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 y keymap) y el número de hilos que usará el compilador

```
> qmk config user.keyboard=elpekenin/access  
> qmk config user.keymap=default  
> qmk config compile.parallel=20  
> qmk config flash.parallel=20
```



4.2.2. Driver para pantalla ili9486

Para hacer pruebas antes de diseñar y fabricar la PCB, he usado un módulo que integra una pantalla, sensor táctil y lector de tarjeta SD.

Sin embargo, QMK no tiene soporte para este modelo concreto de pantalla. Por tanto, usando como base el código[14] para otro dispositivo de la misma familia, y con ayuda de los usuarios @sigprof y @tzarc he desarrollado un driver[15] para la pantalla usada.

Los cambios se reducen a dos cosas:

- Cambiar la fase de inicialización de la pantalla, que configura valores como el formato en el que se envían los píxeles a mostrar
- Puesto que la pantalla contiene un circuito que convierte la línea SPI en una señal paralela de 16 bits que se envía a la pantalla, mediante registros de desplazamiento, tendremos problemas al enviar información de un tamaño que no sea múltiplo de 16, por tanto hacemos un par de cambios al código “normal”.

4.2.3. Dibujar por USB

Lo siguiente que necesitamos resolver es controlar la pantalla desde el ordenador, de forma que podamos dibujar en ella desde nuestro software, para esto usaremos la funcionalidad XAP[16] (aún en desarrollo), que nos proporciona un nuevo endpoint HID[17] sobre el bus USB y un protocolo[18] que podemos extender. Ahora podemos definir nuestros mensajes simplemente creando un archivo `xap.json`[19]

Una vez creados los mensajes, añadimos el código[qp-xap-handler] que se ejecuta cuando los recibamos

4.2.4. Mejora algoritmo elipses

Durante las pruebas con el script anterior, vi que el algoritmo para dibujar elipses funcionaba regular, por lo que hice un *refactor* del mismo. Es decir, mantenemos la misma “forma” (atributos que se reciben y valor devuelto), pero la implementación[20] es completamente diferente. Sigue sin ser perfecto, pero parece funcionar mejor.

4.2.5. Driver para sensor XPT2046

Al igual que hemos necesitado un driver (código que nos permite comunicarnos con el hardware) para dibujar en la pantalla, necesitamos otro para poder leer la posición en la que esta ha sido pulsada. Por desgracia, QMK no tiene ninguna característica parecida, por lo que usaremos el código[21] que he escrito para ello desde 0.

Este hardware, al igual que la pantalla, también tiene una particularidad, y es que la función `spi_receive()` proporcionada por QMK, realmente es un intercambio de información en vez de solo lectura, esto es debido a que SPI es un protocolo síncrono.

El problema aparece porque QMK envía todos los bits a 1 -dado que algunos microcontroladores hacen esto por hardware y no se puede cambiar- para poder recibir información, **sin embargo**, el XPT2046 (a diferencia de la mayoría de chips) utiliza esta información que recibe mientras que nos reporta su información para cambiar su configuración. De forma, que en caso de que los 2 últimos bits que enviamos no tengan el valor adecuado, no podremos usar el sensor. Lo bueno es que la solución es tan simple como usar `spi_write(0)` para enviar un byte vacío y recibir la respuesta dejando una configuración conveniente.



4.2.6. Generar imágenes

Dado que vamos a usar varios iconos para hacer la interfaz gráfica en la pantalla, he cogido la colección de iconos Material Design Icons[22] y he convertido todos sus iconos al formato que utiliza QMK para representar imágenes. Las imágenes originales, scripts usados para la conversión y los archivos en formato QGF[23] se pueden encontrar en este repositorio[24]

4.2.7. Control con una mano

Para dotar de accesibilidad al teclado, vamos a incorporar una funcionalidad opcional que permita que los LEDs debajo de cada tecla se queden apagados, excepto uno que será usado como “selector”. Esta animación se define con el archivo `rgb_matrix_kb.inc??`, que utiliza una variable global para controlar la dirección en la que nos movemos. Al ser una variable global, la podremos editar en el código de usuario (lo que en QMK se conoce como *keymap*), de forma que cada persona pueda añadir *triggers* acorde a sus necesidades, usando joysticks, touchpads, un conjunto de unas pocas teclas, ...

Hay que recordar que lo que acabamos de hacer es solo una animación, es decir, solo sirve para cambiar el estado de los LED bajo cada tecla. Procedemos por tanto a modificar el código que escanea el estado de cada tecla, de forma que podamos modificar su estado acorde a otro trigger, por ejemplo, con un joystick podríamos usar las 4 direcciones para mover el selector y su pulsación para pulsar virtualmente la tecla seleccionada.

4.2.8. Añadir nuestro código

Para poder usar nuestros nuevos archivos, no es suficiente con añadir un `#include` sino que también debemos indicarle a las herramientas de compilado que “busquen” archivos en la carpeta donde tenemos el código, en este caso la carpeta base del teclado, así como subcarpeta `code`. También es necesario, no tengo muy claro por qué, añadir los archivos `c` de las imágenes QGF o no podremos usarlos. Este archivo se complica un poco porque no debemos incluir algunos archivos si no están algunas opciones habilitadas, por ejemplo, no debemos añadir las imágenes si no tenemos la opción de usar la pantalla habilitada. Esto lo hacemos con el archivo `rules.mk`[25]

4.2.9. Comunicación con el software de PC

La idea principal del control de la pantalla es que el teclado no dibuje en ella nada, de forma que la controlaremos exclusivamente desde el programa desarrollado. De esta manera, lo que haremos será leer el estado del sensor táctil cada 200ms y, en caso de estar pulsado, enviaremos las coordenadas de dicha pulsación. Adicionalmente enviamos otro mensaje como si se hubiera pulsado la posición (0, 0), donde no hay ninguna lógica, para que se “limpie” la pantalla al acabar una pulsación. El código relevante es:

```
</> access/keymaps/default/keymap.c

#ifndef TOUCHSCREEN_H
#define TOUCHSCREEN_H

#include "common.h"
#include "matrix.h"

// Variables globales
uint32_t touch_timer = 0;
bool touch_pressed = false;
int16_t touch_x = 0;
int16_t touch_y = 0;

// Función para manejar el sensor táctil
void touch_task(void) {
    // Leer el estado del sensor táctil
    touch_x = touch_get_x();
    touch_y = touch_get_y();

    // Si el sensor táctil detecta un pulso
    if (touch_x > 0 && touch_y > 0) {
        touch_pressed = true;
        touch_timer = 0;
    } else {
        touch_pressed = false;
    }

    // Si el pulso ha durado más de 200ms
    if (touch_timer > 200) {
        touch_pressed = false;
    }
}

// Función para enviar datos al software de PC
void send_touch_data(void) {
    // Envío de datos
    // ...
}

#endif
```



```
if (timer_elapsed32(touch_timer) < 200) // Con un timer controlamos la
frecuencia de muestreo
    return;
touch_timer = timer_read32();

touch_report_t touch_report = get_spi_touch_report(touch_device); // Leemos el sensor
#if defined(XAP_ENABLE) // Si XAP activo, enviamos las coordenadas al
PC
    static bool release_notified = true;
    if (touch_report.pressed) {
        uint8_t payload[4] = { touch_report.x & 0xFF, touch_report.x >> 8,
        touch_report.y & 0xFF, touch_report.y >> 8 };
        // El identificador 0x03 significa que es un mensaje del usuario
        xap_broadcast(0x03, payload, sizeof(payload));

        release_notified = false;
    }
    else if (!release_notified) {
        // Mandamos un mensaje "falso" para limpiar la interfaz
        uint8_t payload[4] = { 0, 0, 0, 0 };
        xap_broadcast(0x03, payload, sizeof(payload));

        release_notified = true;
    }
#endif // XAP_ENABLE
}
#endif // TOUCH_SCREEN
```

Al igual que hemos dicho antes con añadir o no añadir un archivo a nuestro código según la configuración, debemos hacer lo mismo con bloques de código, usando `#ifdef` de forma que no intentemos usar la pantalla táctil o enviar un mensaje XAP si estas características no se han activado.

4.3. Software

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 en Tauri[26] (librería escrita en Rust) ya que permite usar el mismo código en multitud de sistemas operativos gracias a que funciona internamente con un servidor HTML.

4.3.1. Instalación

Para instalar Rust podemos usar pacman `> sudo pacman -S rust`, sin embargo para desarrollar código es preferible usar un script que nos proporciona la comunidad del lenguaje, y que permite cambiar fácilmente la versión del lenguaje con la que compilamos, tan sólo necesitamos ejecutar

```
> curl --proto '=https' --tlsv1.3 -sSf https://sh.rustup.rs -o rust.sh
```

La comunicación con el teclado se realiza usando la librería hidapi[27], a la que accedemos desde Rust gracias al wrapper[28] que implementa una “pasarela” a la librería en C. Para instalarla tan solo necesitamos



añadirla al archivo `Cargo.toml` de nuestro proyecto

Primero instalamos NodeJS con `> sudo pacman -S nodejs`, después clonamos el repositorio `[karl-xap]`, usado de base (y en el que he colaborado) para desarrollar el software.

Para instalar las dependencias de JavaScript ejecutamos `> npm i`. Ahora ya podemos correr `> cargo tauri dev` para lanzar nuestro programa.

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

4.3.2. Desarrollo

Escuchar nuestros mensajes

Lo primero que vamos a añadir es la capacidad de poder recibir los mensajes que nos llegan desde el teclado, para ello definimos un struct que indique el formato del mensaje, podemos ver algo de código que no es necesario entender, lo importante son los dos `u16`

```
</> xap-specs/src/protocol/broadcast.rs

#[derive(BinRead, Debug)]
pub struct ReceivedUserBroadcast {
    pub x: u16,
    pub y: u16,
}

impl XAPBroadcast for ReceivedUserBroadcast {}
```



Acto seguido, hacemos que el *eventloop* de nuestra aplicación escuche los mensajes, ya que por defecto los ignora. Primero definimos un nuevo evento dentro del listado de tipos de evento

```
</> src-tauri/src/events.rs

pub(crate) enum XAPEvent {
    HandleUserBroadcast {
        broadcast: BroadcastRaw,
        id: Uuid,
    },
    // -- Otros eventos recortados --
}
```

Añadimos el código necesario para publicar este nuevo tipo de evento al recibir los mensajes correspondientes

```
</> src-tauri/src/xap/hid/device.rs

match broadcast.broadcast_type() {
    // -- Otros tipos recortados --
    BroadcastType::User => {
        event_channel
            .send(XAPEvent::ReceivedUserBroadcast { id, broadcast })
            .expect("failed to send user broadcast event!");
    }
}
```

Y por último hacemos la relación entre este evento la lógica de usuario que se debe ejecutar con ellos.

```
</> src-tauri/src/main.rs

match broadcast.broadcast_type() {
    // -- Otros eventos recortados --
    Ok(XAPEvent::ReceivedUserBroadcast{broadcast, id}) => {
        user::broadcast_callback(broadcast, id, &state);
    }
}
```

Toda la lógica de usuario (personalizable), se puede encontrar en el archivo `src-tauri/src/user.rs`[29]



Correr aplicación en segundo plano

Dado que todo el contenido de la pantalla, así como su funcionalidad al pulsarla dependen de nuestra aplicación, vamos a hacer todo lo posible para que se mantenga abierta. Para ello hacemos que al pulsar el botón de cerrar, la app se minimice en vez de terminar y le añadimos un *systray* para tener un ícono en la barra de tareas donde podamos volver a ponerla en pantalla

```
</> src-tauri/src/main.rs

// Creamos el systray
let tray_menu = SystemTrayMenu::new()
    .add_item(CustomMenuItem::new("show".to_string(), "Show"))
    .add_item(CustomMenuItem::new("hide".to_string(), "Hide"))
    .add_native_item(SystemTrayMenuItem::Separator)
    .add_item(CustomMenuItem::new("quit".to_string(), "Quit"));

// Lo incluimos en el programa, así como su lógica
.system_tray(system_tray)
.on_system_tray_event(move |app, event|
    match event {
        // Al usar click izquierdo
        SystemTrayEvent::MenuItemClick { id, .. } => {
            // Según botón usado
            match id.as_str() {
                "hide" => app.get_window("main").unwrap().hide().unwrap(),
                "quit" => {
                    user::on_close(_state.clone());
                    std::process::exit(0);
                },
                "show" => app.get_window("main").unwrap().show().unwrap(),
                _ => {}
            }
        },
        // Otros eventos, no hacemos nada
        _ => {}
    }
)

// Interceptamos el cierre para evitarlo
.on_window_event(|event| match event.event() {
    tauri::WindowEvent::CloseRequested { api, .. } => {
        event.window().hide().unwrap();
        api.prevent_close()
    },
    // Otros eventos, no hacemos nada
    _ => {}
})
```



Indicador de conexión

Vamos a añadir también un indicador en la pantalla, de forma que Tauri nos haga saber cuando se conecta o desconecta del teclado, evitando que intentemos usar la pantalla cuando no va a funcionar. Editamos el eventloop y definimos las nuevas funciones

```
</> src-tauri/src/main.rs

match msg {
    Ok(XAPEvent::Exit) => {
        info!("received shutdown signal, exiting!");
        + user::on_close(state);
        break 'event_loop;
    },
    Ok(XAPEvent::NewDevice(id)) => {
        if let Ok(device) = state.lock().get_device(&id){
            info!("detected new device - notifying frontend!");
            + user::on_device_connection(device);
    }
}
```

```
</> src-tauri/src/user.rs

pub(crate) fn on_device_connection(device: &XAPDevice) {
    // Necesitamos esperar un poco para que el teclado pueda escucharnos
    std::thread::sleep(std::time::Duration::from_millis(280));

    // Show connection
    let _ = device.query(PainterDrawTextRecolor(
        PainterTextRecolor {
            dev: SCREEN_ID,
            x: 15,
            y: 15,
            font: 0,
            fg_color: FG_COLOR,
            bg_color: BG_COLOR,
            text: "Connected to Tauri".into(),
        }
    ));

    // Print buttons
    for id in 0..N_BUTTONS {
        let _ = device.query(PainterDrawImageRecolor (
            PainterImageRecolor {
                dev: SCREEN_ID,
                x: BUTTONS_X[id],
                y: BUTTONS_Y[id],
                img: BUTTON2IMG[id],
                fg_color: FG_COLOR,
                bg_color: BG_COLOR,
            }
        )));
    }
}
```



```
pub(crate) fn on_close(state: Arc<Mutex<XAPClient>>) {
    for device in state.clone().lock().get_devices() {
        // Clear screen
        let _ = device.query(PainterDrawRect {
            PainterRect {
                dev: SCREEN_ID,
                left: 0,
                top: 0,
                right: SCREEN_WIDTH,
                bottom: SCREEN_HEIGHT,
                color: BG_COLOR,
                filled: 1
            }
        });
        // Show text
        let _ = device.query(PainterDrawTextRecolor(
            PainterTextRecolor {
                dev: SCREEN_ID,
                x: 15,
                y: 15,
                font: 0,
                fg_color: FG_COLOR,
                bg_color: BG_COLOR,
                text: "Tauri app was closed".into(),
            }
        ));
    };
}
```

Arrancar HomeAssistant

Dado que nuestro programa se tiene que comunicar con la domótica de casa, vamos a asegurarnos de que esté ejecutándose, iniciándolo al abrir el programa. Al inicio del archivo `main.rs` ejecutamos `user::on_init()`:

```
</> src-tauri/src/user.rs

pub(crate) fn on_init() {
    match std::process::Command::new("sh")
        .arg("-c")
        .arg("sudo systemctl start docker && cd $HOME/docker && docker compose up -d")
    {
        .output()
    {
        Ok(_) => error!("on_init went correctly"),
        Err(out) => error!("on_init failed due to: {out}")
    }
}
```

SECCIÓN 5

ANEXO I: INSTALACIÓN DE MICROPYTHON

Durante el desarrollo del proyecto, he probado MicroPython[30], 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++.

5.1. 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.

5.2. 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
```

5.3. 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) [31]

Definimos en C el módulo `usb_hid`[32], será `MP_REGISTER_MODULE` la macro encargada de añadir el módulo al firmware compilado. También debemos editar un archivo de configuración de compilación para que se añada el nuevo archivo `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 en Arch he necesitado instalar el paquete

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



Por último, compilamos con

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

Y ya podremos flashear este binario en nuestra placa de desarrollo

SECCIÓN 6

REFERENCIAS

1. *Home Assistant, gestor de domótica*, <https://www.home-assistant.io/>.
2. *Windows Subsystem for Linux (WSL)*, <https://learn.microsoft.com/es-es/windows/wsl/about>.
3. *VcXsrv, servidor X de gráficos para Windows*, <https://sourceforge.net/projects/vcxsvr/>.
4. *Función en ZSH para compilar LATEX*, https://github.com/elpekenin/access_kb/blob/main/utils/pdf.sh.
5. *Estudios médicos sobre diseño de teclados*, <https://pubmed.ncbi.nlm.nih.gov/19308823/>.
6. tzarc, *Teclado ghoul*, <https://github.com/tzarc/ghoul>.
7. *Protocolo SPI*, https://es.wikipedia.org/wiki/Serial_Peripheral_Interface.
8. *Protocolo I2C*, <https://es.wikipedia.org/wiki/I%C2%B2C>.
9. *Quantum Painter, API para GUI*, https://docs.qmk.fm/#/quantum_painter.
10. *Pointing Device, sensores de movimiento*, https://docs.qmk.fm/#/feature_pointing_device.
11. *QMK, librería para desarrollar firmware de teclados*, https://github.com/Jpe230/qmk_firmware.
12. *LVGL, librería para gráficos en sistemas embedidos*, <https://lvgl.io>.
13. jpe230, *Añadir LVGL en QMK*, https://github.com/Jpe230/qmk_firmware/commits/develop_lvgl.
14. *Driver para ili9488*, https://github.com/qmk/qmk_firmware/blob/master/drivers/painter/ili9xxx/qp_ili9488.c.
15. *Mi código para ili9486*, https://github.com/qmk/qmk_firmware/pull/18521.
16. *XAP, protocolo extensible de QMK*, https://github.com/qmk/qmk_firmware/projects/4.
17. *Protocolo HID*, <https://es.wikipedia.org/wiki/HID>.
18. *Especificación del protocolo XAP*, https://github.com/qmk/qmk_firmware/blob/xap/docs/xap_protocol.md.
19. *Mensajes personalizados en XAP*, https://github.com/elpekenin/qmk_firmware/blob/pekelop_users/elpekenin/xap.json.
20. *Nuevo algoritmo para dibujar elipses*, https://github.com/qmk/qmk_firmware/pull/19005.
21. *Driver para el sensor táctil XPT2046*, https://github.com/elpekenin/qmk_firmware/blob/pekelop/users/elpekenin/touch.
22. *Material Design Icons*, <https://github.com/Templarian/MaterialDesign>.
23. *Formato de imágenes para QMK*, https://docs.qmk.fm/#/quantum_painter_qgf.
24. *Repositorio con iconos en QGF*, <https://github.com/elpekenin mdi-icons-qgf>.
25. *Añadir nuestro código a la compilación de QMK*, https://github.com/elpekenin/qmk_firmware/blob/pekelop/users/elpekenin/rules.mk.



26. *Tauri, librería para hacer aplicaciones multi-plataforma*, <https://tauri.app/>.
27. *Librería hidapi (C)*, <https://github.com/libusb/hidapi>.
28. *Librería para usar hidapi en Rust*, <https://docs.rs/hidapi/latest/hidapi/>.
29. *Código de usuario*, https://github.com/elpekenin/qmk_xap/blob/peke-devel/src-tauri/src/user.rs.
30. *MicroPython, implementacion de Python para MCU*, <https://micropython.org/>.
31. noobee, *Añadir HID a RP2040 en MicroPython*, <https://github.com/noobee/micropython/tree/usb-hid>.
32. *Módulo para usar HID en MicroPython*, https://github.com/elpekenin/micropython/blob/peke-devel/ports/rp2/modusb_hid.c.
33. *Formatear Rust en LATEX*, <https://github.com/denki/listings-rust>.