

**Universidad
Politécnica
de Cartagena**

Revisitando el diseño del teclado

Autor: Pablo Martínez Bernal

Director: José Alfonso Vera Repullo

Máster Universitario en Ingeniería de Telecomunicación

Feb-2024

Autor: Pablo Martínez Bernal

email: elpekenin@elpekenin.dev

Índice

1. Listado de acrónimos	3
2. Resumen	7
3. Introducción	8
3.1. Motivación	8
3.2. Objetivo	8
3.3. Requisitos previos	9
4. Estado del arte	10
4.1. Diseño	10
4.1.1. Diseño anticuado	10
4.1.1.1. Forma del teclado	10
4.1.1.2. Ubicación de las letras	12
4.1.2. Ergonomía	12
4.2. Hardware	14
4.2.1. Cableado de las teclas	14
4.2.1.1. Conexión directa	15
4.2.1.2. Matriz	15
4.2.1.3. Lectura en serie	17
4.2.2. Pantallas	18
4.2.3. Sensor táctil	18
4.3. Firmware	18
4.3.1. Funcionalidad	18
4.3.2. Escaneo de teclas	19
4.3.3. Pantallas	19
4.3.4. Pantalla táctil	19
5. Desarrollo	20
5.1. Hardware	20
5.1.1. Objetivos	20
5.1.2. Distribución	20
5.1.3. Desarrollo	20
5.1.3.1. Teclas	20
5.1.3.2. Cableado de las pantallas	22
5.2. Firmware	23
5.2.1. Instalación y configuración de QMK	23

5.2.1.1. Errores de compilación	23
5.2.2. Driver para pantalla ili9486	24
5.2.3. Dibujar por USB	25
5.2.4. Mejora algoritmo elipses	25
5.2.5. Driver para sensor XPT2046	25
5.2.6. Generar imágenes	26
5.2.7. Añadir nuestro código	26
5.2.8. Comunicación con el software de PC	26
5.3. Software	27
5.3.1. Instalación	27
5.3.2. Desarrollo	27
5.3.2.1. Escuchar nuestros mensajes	28
5.3.2.2. Correr aplicación en segundo plano .	29
5.3.2.3. Indicador de conexión	30
5.3.2.4. Arrancar HomeAssistant	31
6. Lineas futuras	32
7. Anexo I: Instalación de MicroPython	33
7.1. Preparar el compilador	33
7.2. Compilar para Linux (Opcional)	33
7.3. Compilar para RP2040	33
8. Bibliografía	35
9. Anexo B. Código fuente del informe (typst).	37

1. Listado de acrónimos

HID:	Human Interface Device
I2C:	Inter Integrated Circuit
MCU:	Micro Controller Unit
PCB:	Printed Circuit Board
SPI:	Serial Peripheral Interface
USB:	Universal Serial Bus

Listado de imágenes

Figura 1: Comparativa teclado antiguo/actual	10
Figura 2: Detalle de las letras en una máquina de escribir	11
Figura 3: «Mapa» de mecanografía	11
Figura 4: Teclado ortolineal	11
Figura 5: Comparativa distribuciones	12
Figura 6: Teclado <i>Quefrency</i>	13
Figura 7: Teclado <i>Dygma Raise</i>	13
Figura 8: Teclado <i>Moonlander</i>	13
Figura 9: Teclado <i>Ergodox</i>	14
Figura 10: Teclado <i>Dactyl Manuform</i> con trackball	14
Figura 11: Cableado directo	15
Figura 12: Cableado en matriz	16
Figura 13: Ghosting en una matriz	16
Figura 14: Matriz anti-ghosting	17
Figura 15: Diseño aproximado del teclado	20
Figura 16: MOSFET como inversor	21
Figura 17: SN74HC589ADR2G para una fila	22

Listado de código

Código 1: Definición de mensaje de usuario	28
Código 2: Definición de nuevo evento	28
Código 3: Emitir evento al recibir mensaje	28
Código 4: Manejo del evento	29
Código 5: Creación del <i>systray</i>	29
Código 6: Añadir <i>systray</i> al programa, con su logica	30
Código 7: Interceptar señal de cierre de ventana	30
Código 8: Hooks de usuario	31
Código 9: Hook de inicio	31

Listado de comandos

Comando 1: Downgrade de GCC	23
Comando 2: Personalizar QMK	24
Comando 3: Compilar MicroPython para Linux	33
Comando 4: Ejecutar MicroPython en Linux	33
Comando 5: Compilar MicroPython para RP2040	34

2. 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.

3. Introducción

3.1. 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.

3.2. 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 el 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.

3.3. Requisitos previos

A lo largo del documento se asume cierto conocimiento de programación, así como tener algunos programas instalados. La mayoría son conocidos, pero aquí se puede encontrar una lista con algunos menos extendidos, así como enlaces a sus respectivas páginas oficiales/documentación

- Rust [2]. Lenguaje de programación
- typst [3]. Lenguaje de marcado, parecido a LaTex, con el que se ha compuesto este documento.
- WSL [4]. Herramienta de Windows, similar a una máquina virtual, para tener un entorno Linux

4. Estado del arte

4.1. Diseño



IBM Model M (1984)



Logitech MX Mechanical (2022)



Teclado Android

Figura 1: Comparativa teclado antiguo/actual

Como podemos ver, el panorama apenas ha variado en los últimos 40 años, ni siquiera para adaptarse a las pequeñas pantallas táctiles de los móviles.

4.1.1. Diseño anticuado

4.1.1.1. 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 2: 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 3: «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%»

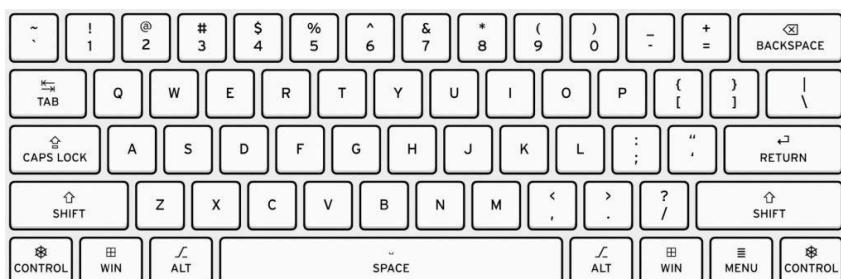


Figura 4: Teclado ortolineal

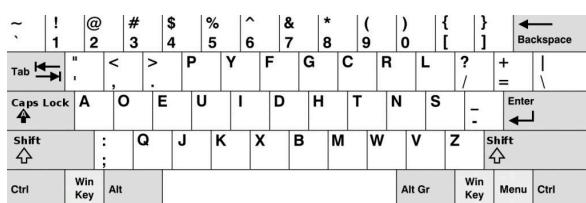
4.1.1.2. Ubicación de las letras

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.

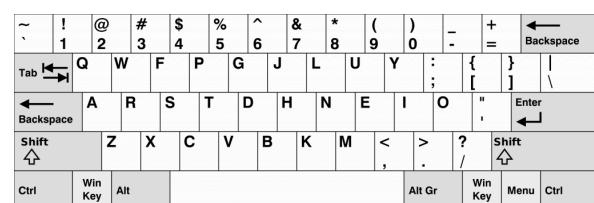
En contra, ahora que gracias a la electrónica no tenemos estas limitaciones, se han diseñado distribuciones que minimizan la distancia media que se debe recorrer al escribir, por lo que una vez acostumbrados a ellas se puede escribir más rápido y reduciendo la fatiga en los dedos. Las dos más extendidas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



QWERTY



DVORAK



COLEMAK

Figura 5: Comparativa distribuciones

4.1.2. 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 6: 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 levantarla.



Figura 7: Teclado *Dygma Raise*

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



Figura 8: 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 comunmente se conoce como *thumb cluster*.



Figura 9: 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 10: 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

4.2. Hardware

4.2.1. Cableado de las teclas

4.2.1.1. 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:

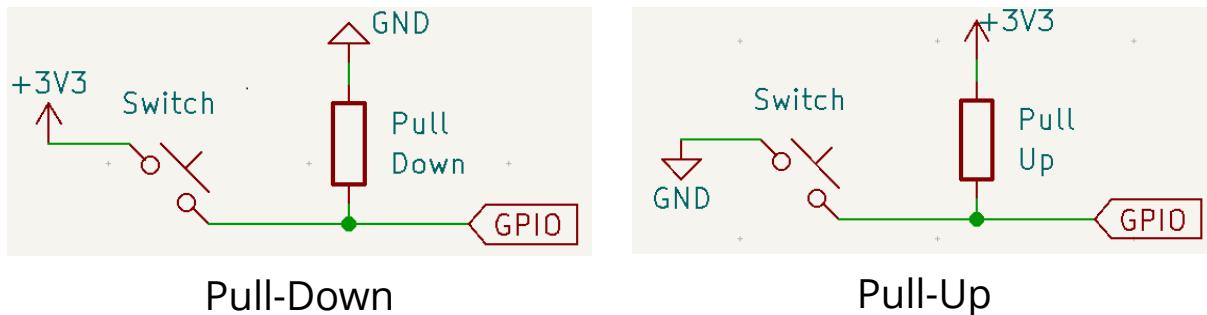


Figura 11: Cableado directo

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.

4.2.1.2. 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

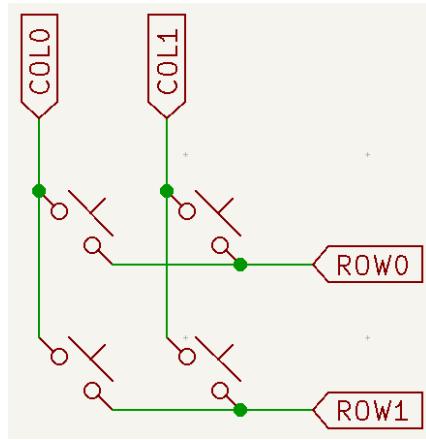


Figura 12: Cableado en matriz

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.

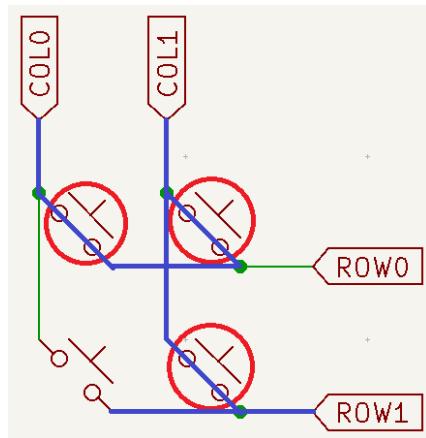


Figura 13: Ghosting en una matriz

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.

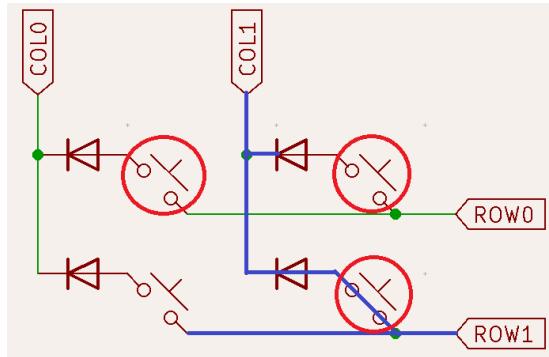


Figura 14: Matriz anti-ghosting

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 matrix 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.

4.2.1.3. 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)

4.2.2. 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 pequeños dibujos en estas pantallas más comunes.

4.2.3. Sensor táctil

También es relativamente común encontrar diseños que incluyen diferentes sensores (por ejemplo joysticks analógicos o PMW3360) para mover el sensor por la pantalla del ordenador sin tener que mover la mano hasta el ratón. Tal como podemos ver en el Dactyl[Figura 10]

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.

4.3. Firmware

4.3.1. 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 Universal Serial Bus (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 micro-controladores que internamente utilizan código y hardware muy diferente
- Documentación extensa y detallada
- Servidor oficial en Discord donde podemos encontrar ayuda

4.3.2. 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 *I/O 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.

4.3.3. 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.

4.3.4. 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.

5. Desarrollo

5.1. Hardware

5.1.1. 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.

5.1.2. Distribución

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



Figura 15: Diseño aproximado del teclado

5.1.3. Desarrollo

5.1.3.1. Teclas

Como ya vimos en la sección 3.2.1 Sección 4.2.1, el escaneo de teclas se hará mediante registros de desplazamiento de entrada

en paralelo y salida en serie, para un mejor tiempo de respuesta vamos a usar componentes SPI en vez de I2C (protocolo más rápido). Podríamos haber usado SN74HC165 que son algo más baratos y fáciles de encontrar, pero vamos a usar SN74HC589ADR2G ya que tienen el mismo funcionamiento pero presentan 7 de las 8 entradas en el mismo lado del circuito integrado (en vez de 4 en cada lado), por lo que son mucho más sencillos de enrutar. Además, aunque cada mitad del teclado tiene un total de 29 teclas y, por tanto, 4 registros (32 entradas) nos hubieran bastado; vamos a usar un registro por cada fila (5), ya que evitamos tener que enrutar una de las teclas hacia el otro lado del registro y tener algunas conexiones «saltando» de una fila a otra.

Realizamos las conexiones de la siguiente manera:

- Las entradas que no usamos las conectaremos a tierra para que se lean como “0”. Las teclas se conectan a VCC y (mediante «pull-down») a una entrada. De esta forma leeremos “1” cuando estén pulsadas y “0” cuando no lo estén
- La salida serie de cada integrado se conecta a la entrada serie del siguiente. El último se conecta a MISO (entrada SPI del microcontrolador)
- Las señales CS, Latch y CLK son comunes a todos los integrados
- Como la señal Latch resulta ser la inversa de CS, en vez de usar otro pin, invertimos el voltaje con un MOSFET

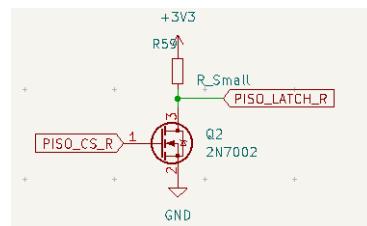


Figura 16: MOSFET como inversor

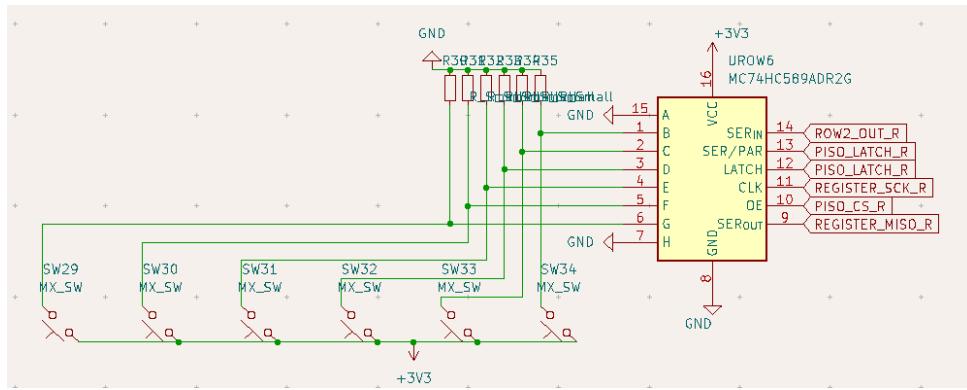


Figura 17: SN74HC589ADR2G para una fila

5.1.3.2. Cableado de las pantallas

Vamos a usar una pantalla en el lado izquierdo (IL91874) y dos en el derecho (ILI9163 e ILI9341), estos dispositivos necesitan de conexión SPI así como pines extra (DC, CS, RST) para ser controlados. Dado que las vamos a conectar al mismo bus, solo necesitamos 3 pines para SPI y puesto que no podemos mandar información a dos pantallas simultáneamente, también pueden usar una línea DC común. Sin embargo las señales de DC y RST deben ser individuales y además la ILI9341 necesita dos señales DC, una para la pantalla y otra para el sensor táctil. Por tanto, también usaremos registros de desplazamiento, en este caso de entrada serie con salidas en paralelo (SN74HC595) y dichas salidas se conectarán a las entradas de control de las pantallas. Como ya teníamos un bus SPI configurado, usar estos registros tan solo supone el uso de otro pin (CS). Sin embargo, como necesitamos cambiar estas señales mientras estamos hablando con las pantallas, no podemos tener las dos cosas en el mismo bus, por lo que usaremos un bus para los registros (escanear teclas y señales de control) y otro bus para las pantallas. Lo más importante de este diseño es que encadenando suficientes registros podríamos controlar tantas señales de control para diversos dispositivos como queramos, manteniendo mínima la utilización de pines del MCU.

5.2. Firmware

5.2.1. Instalación y configuración de QMK

Para usar QMK[11] instalamos su CLI ejecutando 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 , a este comando le podemos pasar como parámetro nuestro fork del repositorio (en mi caso), 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 en . Finalmente podemos ejecutar para comprobar el estado de QMK. Para poder hacer debug con he necesitado añadir una línea a dicho archivo:

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 (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**

5.2.1.1. Errores de compilación

Es posible que al intentar compilar obtengamos un error parecido a, 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:

```
elpekenin@PC:~$ 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
```

Comando 1: Downgrade de GCC

Una vez configurado QMK, creamos los archivos básicos para el firmware de nuestro teclado haciendo e introduciendo los datos necesarios, sin embargo esto crea una carpeta directamente en la ruta , 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

```
elpekenin@PC:~$ qmk config user.keyboard=elpekenin/access  
elpekenin@PC:~$ qmk config user.keymap=default  
elpekenin@PC:~$ qmk config compile.parallel=20
```

Comando 2: Personalizar QMK

5.2.2. 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».

5.2.3. 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 [19] Una vez creados los mensajes, añadimos el código[20] que se ejecuta cuando los recibamos

5.2.4. 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[21] es completamente diferente. Sigue sin ser perfecto, pero parece funcionar mejor.

5.2.5. 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[22] que he escrito para ello desde 0.

Este hardware, al igual que la pantalla, también tiene una particularidad, y es que la función 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 configura-

ción. 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 para enviar un byte vacío y recibir la respuesta dejando una configuración conveniente.

5.2.6. 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[23] 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[24] se pueden encontrar en este repositorio[25]

5.2.7. Añadir nuestro código

Para poder usar nuestros nuevos archivos, no es suficiente con añadir un 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 . También es necesario, no tengo muy claro por qué, añadir los archivos 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 [26]

5.2.8. 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:

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 de forma que no intentemos usar la pantalla táctil o enviar un mensaje XAP si estas características no se han activado.

5.3. 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[27] (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.

5.3.1. Instalación

Para instalar Rust podemos usar pacman , 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

La comunicación con el teclado se realiza usando la librería hidapi[28], a la que accedemos desde Rust gracias al wrapper[29] que implementa una «pasarela» a la librería en C. Para instalarla tan solo necesitamos añadirla al archivo de nuestro proyecto

Primero instalamos NodeJS con , después clonamos el repositorio[30], usado de base (y en el que he colaborado) para desarrollar el software. Para instalar las dependencias de JavaScript ejecutamos . Ahora ya podemos correr para lanzar nuestro programa.

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

5.3.2. Desarrollo

5.3.2.1. 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

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

Código 1: Definición de mensaje de usuario

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

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

Código 2: Definición de nuevo evento

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

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

Código 3: Emitir evento al recibir mensaje

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

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

Código 4: Manejo del evento

Toda la lógica de usuario (personalizable), se puede encontrar en el archivo [31]

5.3.2.2. 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

```
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"));
```

Código 5: Creación del *systray*

```

    .system_tray(system_tray)
    .on_system_tray_event(move |app, event|
        match event {
            // Al usar click izquierdo
            SystemTrayEvent::MenuItemClick { id, .. } => {
                // Segun boton 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
        }
    )
)

```

Código 6: Añadir *systray* al programa, con su logica

```

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

```

Código 7: Interceptar señal de cierre de ventana

5.3.2.3. 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

```

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);
        }
    }
}

```

Código 8: Hooks de usuario

5.3.2.4. 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 arrancar la aplicación ejecutamos:

```

pub(crate) fn on_init() {
    match std::process::Command::new("sh")
        .arg("-c")
        .arg(r#"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}")
    }
}

```

Código 9: Hook de inicio

6. Lineas futuras

7. Anexo I: Instalación de MicroPython

Durante el desarrollo del proyecto, he probado *MicroPython*[32] 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 , hacemos 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

```
elpekenin@PC:~$ cd ports/unix  
elpekenin@PC:~$ make submodules  
elpekenin@PC:~$ make
```

Comando 3: Compilar MicroPython para Linux

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

Comando 4: Ejecutar MicroPython en Linux

7.3. Compilar para RP2040

Primero instalamos un compilador necesario para la arquitectura del procesador, en mi caso (Arch Linux), el comando es y después

añadimos la configuración necesaria para reportar y usar un endpoint HID, siguiendo (y adaptando) [33]

Definimos en C el módulo [34], será 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

Para poder compilar la versión de RP2040 en Arch he necesitado instalar el paquete

Por último, compilamos con

```
elpekenin@PC:~$ cd ports/rp2
elpekenin@PC:~$ make submodules
elpekenin@PC:~$ make clean
elpekenin@PC:~$ make
```

Comando 5: Compilar MicroPython para RP2040

Y ya podremos flashear este binario en nuestra placa de desarrollo

8. Bibliografía

- [1] «Home Assistant, gestor de domótica».
- [2] «Rust. Lenguaje de programación».
- [3] «typst».
- [4] «Windows Subsystem for Linux (WSL)».
- [5] «Estudios médicos sobre diseño de teclados».
- [6] tzarc, «Teclado ghoul».
- [7] «Protocolo SPI».
- [8] «Protocolo I2C».
- [9] «Quantum Painter, API para GUI».
- [10] «Pointing Device, sensores de movimiento».
- [11] «QMK, librería para desarrollar firmware de teclados».
- [12] «LVGL, librería para gráficos en sistemas embedidos».
- [13] jpe230, «Añadir LVGL en QMK».
- [14] «Driver para ili9488».
- [15] «Mi código para ili9486».
- [16] «XAP, protocolo extensible de QMK».
- [17] «Protocolo HID».
- [18] «Especificación del protocolo XAP».
- [19] «Mensajes personalizados en XAP».
- [20] «Handlers de los mensajes XAP personalizados».
- [21] «Nuevo algoritmo para dibujar elipses».
- [22] «Driver para el sensor táctil XPT2046».
- [23] «Material Design Icons».

- [24] «Formato de imágenes para QMK».
- [25] «Repositorio con iconos en QGF».
- [26] «Añadir nuestro código a la compilación de QMK».
- [27] «Tauri, librería para hacer aplicaciones multi-plataforma».
- [28] «Librería hidapi (C)».
- [29] «Librería para usar hidapi en Rust».
- [30] KarlK90, «qmk_xap».
- [31] «Código de usuario».
- [32] «MicroPython, implementacion de Python para MCU».
- [33] noobee, «Añadir HID a RP2040 en MicroPython».
- [34] «Módulo para usar HID en MicroPython».

9. Anexo B. Código fuente del informe (typst).

Aquí están adjuntas las primeras líneas del código fuente con el que he generado este documento.

```
// 3rd party
#import "@preview/acrostiche:0.3.1": acr, init-acronyms, print-index
#import "@preview/sourcerer:0.2.1"

// Helpers
#import "typ/funcs.typ": cli, snippet, page_footer

// Show
#show heading.where(level: 1): it => { pagebreak() + it }
#show bibliography: set heading(numbering: "1.")

// Set
#set heading(numbering: "1.")
#set page.footer: page_footer
#set par(justify: true)
#set text(font: "Open Sans", lang: "es", ligatures: true, size: 15pt, slashed-zero: true)

// Misc
#init-acronyms(
    "HID": ("Human Interface Device"),
    "I2C": ("Inter Integrated Circuit"),
    "MCU": ("Micro Controller Unit"),
    "PCB": ("Printed Circuit Board"),
    "SPI": ("Serial Peripheral Interface"),
    "USB": ("Universal Serial Bus"),
)

// Content start
#page[
    #align(center + horizon)[
        #image("images/UPCT-front.jpg", width: 70%)
    ]
    #align(center)[
        #text(size: 25pt, weight: "bold") [Revisitando el diseño del teclado]
    ]
    #align(bottom + right)[
        #text(weight: "bold", size: 15pt)[
            Autor: Pablo Martínez Bernal
            Director: José Alfonso Vera Repullo
            Máster Universitario en Ingeniería de Telecomunicación
            #datetime.today().display("[month repr:short]-[year]")
        ]
    ]
    // 0 to not render page number here
    #counter(page).update(0)
]
#page[
    #align(center + horizon)[
        #table(
            columns: 1,
            [*Autor*: Pablo Martínez Bernal],
            [*email*: elpekenin@elpekenin.dev],
        )
    ]
    // 0 to not render page number here
    #counter(page).update(0)
]
```

```

// File content overview
#outline(
    fill: block(width: 100% - 1.5em)[
        #repeat(" " . " ")
    ],
    indent: auto,
)
#outline(
    print-index(
        outlined: true,
        sorted: "up",
        title: [Listado de acrónimos],
    )
)
#outline(
    target: figure.where(kind: image),
    title: [Listado de imágenes],
)
#outline(
    target: figure.where(kind: "snippet"),
    title: [Listado de código],
)
#outline(
    target: figure.where(kind: "cmd"),
    title: [Listado de comandos],
)

= 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.

<chapter:contents>

= Introducción
#include "typ/intro.typ"

== Requisitos previos
A lo largo del documento se asume cierto conocimiento de programación, así como tener algunos programas instalados. La mayoría son conocidos, pero aquí se puede encontrar una lista con algunos menos extendidos, así como enlaces a sus respectivas páginas oficiales/documentación
- Rust @rust. Lenguaje de programación
- typst @typst. Lenguaje de marcado, parecido a LaTex, con el que se ha compuesto este documento.
- WSL @wsl. Herramienta de Windows, similar a una máquina virtual, para tener un entorno Linux

= Estado del arte
#include "typ/state.typ"

== Hardware

==== Cableado de las teclas <sec:scanning>
===== 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:

#figure(
    grid(
        columns: (auto, auto),
        gutter: 1em,
        figure(
            image("images/pull_down.png"),
            caption: [Pull-Down],
            numbering: none,
            outlined: false,
        ),
        figure(

```

```

        image("images/pull_up.png"),
        caption: [Pull-Up],
        numbering: none,
        outlined: false,
    )
),
caption: [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

```

#figure(
    image("images/matrix.png", width: 35%),
    caption: [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.

```

#figure(
    image("images/ghosting.png", width: 35%),
    caption: [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.

```

#figure(
    image("images/anti_ghosting.png", width: 45%),
    caption: [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 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_* #cite(<ghoul>), 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#cite(<spi>) o I2C#cite(<i2c>). 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)

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 pequeños dibujos en estas pantallas más comunes.

Sensor táctil

También es relativamente común encontrar diseños que incluyen diferentes sensores (por ejemplo joysticks analógicos o PMW3360) para mover el sensor por la pantalla del ordenador sin tener que mover la mano hasta el ratón. Tal como podemos ver en el Dactyl[[@img:dactyl](#)]

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.

== Firmware

==== 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 `#acsr("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

==== 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.

==== Pantallas

QMK tiene una API estandarizada (Quantum Painter`#cite(<qp>)`) para primitivas de dibujo en interfaces gráficas. Aún es "joven" y no soporta demasiadas pantallas, pero hace gran parte del trabajo por nosotros.

==== Pantalla táctil

QMK también tiene una capa de abstracción`#cite(<pointing>)` 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.

= Desarrollo

== Hardware

==== 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.

==== Distribución

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

```
#figure(  
    image("images/layout.png", width: 100%),  
    caption: [Diseño aproximado del teclado]  
)
```

==== Desarrollo

===== Teclas

Como ya vimos en la sección 3.2.1 `@sec:scanning`, el escaneo de teclas se hará mediante registros de desplazamiento de entrada en paralelo y salida en serie, para un mejor tiempo de respuesta vamos a usar componentes SPI en vez de I2C (protocolo más rápido). Podríamos haber usado SN74HC165 que son algo más baratos y fáciles de encontrar, pero vamos a usar SN74HC589ADR2G ya que tienen el mismo funcionamiento pero presentan 7 de las 8 entradas en el mismo lateral del circuito integrado (en vez de 4 en cada lado), por lo que son mucho más sencillos de enrutar. Además, aunque cada mitad del teclado tiene un total de 29 teclas y, por tanto, 4 registros (32 entradas) nos hubieran bastado; vamos a usar un registro por cada fila (5), ya que evitamos tener que enrutar una de las teclas hacia el otro lado del registro y tener algunas conexiones "saltando" de una fila a otra.

Realizamos las conexiones de la siguiente manera:

- Las entradas que no usamos las conectaremos a tierra para que se lean como '0'. Las teclas se conectan a VCC y (mediante "pulldown") a una entrada. De esta forma leeremos '1' cuando estén pulsadas y '0' cuando no lo estén
- La salida serie de cada integrado se conecta a la entrada serie del siguiente. El último se conecta a MISO (entrada SPI del microcontrolador)
- Las señales CS, Latch y CLK son comunes a todos los integrados
- Como la señal Latch resulta ser la inversa de CS, en vez de usar otro pin, invertimos el voltaje con un MOSFET

```

#figure(
    image("images/inverter.png", width: 30%),
    caption: [MOSFET como inversor]
)

#figure(
    image("images/piso.png", width: 80%),
    caption: [SN74HC589ADR2G para una fila]
)

==== Cableado de las pantallas
Vamos a usar una pantalla en el lado izquierdo (ILI91874) y dos en el derecho (ILI9163 e ILI9341), estos dispositivos necesitan de conexión SPI así como pines extra (DC, CS, RST) para ser controlados. Dado que las vamos a conectar al mismo bus, solo necesitamos 3 pines para SPI y puesto que no podemos mandar información a dos pantallas simultáneamente, también pueden usar una línea DC común. Sin embargo las señales de DC y RST deben ser individuales y además la ILI9341 necesita dos señales DC, una para la pantalla y otra para el sensor táctil. Por tanto, también usaremos registros de desplazamiento, en este caso de entrada serie con salidas en paralelo (SN74HC595) y dichas salidas se conectarán a las entradas de control de las pantallas. Como ya teníamos un bus SPI configurado, usar estos registros tan solo supone el uso de otro pin (CS). Sin embargo, como necesitamos cambiar estas señales mientras estamos hablando con las pantallas, no podemos tener las dos cosas en el mismo bus, por lo que usaremos un bus para los registros (escanear teclas y señales de control) y otro bus para las pantallas. Lo mas importante de este diseño es que encadenando suficientes registros podríamos controlar tantas señales de control para diversos dispositivos como queramos, manteniendo mínima la utilización de pines del MCU.

== Firmware

==== Instalación y configuración de QMK
Para usar QMK#cite(<qmk>) instalamos su CLI ejecutando 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 , a este comando le podemos pasar como parámetro nuestro fork del repositorio (en mi caso ), 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 en . Finalmente podemos ejecutar para comprobar el estado de QMK. Para poder hacer debug con he necesitado añadir una línea a dicho archivo:

Opcionalmente, podríamos instalar LVGL#cite(<lvgl>) funciones gráficas más complejas en la pantalla LCD, en vez de usar el driver de QMK. Esto se hace con (desde el directorio base de QMK). Seguidamente, usamos el código de jpe#cite(<lvgl-jpe>) para usar esta librería.

De momento no he implementado esta librería puesto que añadiría bastante complejidad a *#link(<section:dibujar-usb>)[4.2.3]*

==== Errores de compilación
Es posible que al intentar compilar obtengamos un error parecido a, 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:
#cli(
    ````bash
 $ 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
    ```,
    caption: [Downgrade de GCC]
)

Una vez configurado QMK, creamos los archivos básicos para el firmware de nuestro teclado haciendo e introduciendo los datos necesarios, sin embargo esto crea una carpeta directamente en la ruta , 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

#cli(
    ````bash
 $ qmk config user.keyboard=elpekenin/access
 $ qmk config user.keymap=default
 $ qmk config compile.parallel=20
    ```,
    caption: [Personalizar QMK]
)

==== 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#cite(<ili9488>) para otro dispositivo de la misma familia, y con ayuda de los usuarios \sigprof y \tzarc he desarrollado un driver#cite(<ili9486-pr>) 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".

==== Dibujar por USB <section:dibujar-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#cite(<xap>) (aún en desarrollo), que nos proporciona un nuevo endpoint HID#cite(<hid>) sobre el bus USB y un protocolo#cite(<xap-specs>) que podemos extender. Ahora podemos definir nuestros mensajes simplemente creando un archivo #cite(<xap-custom-msg>) Una vez creados los mensajes, añadimos el código#cite(<qp-xap-handlers>) que se ejecuta cuando los recibamos

==== 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#cite(<ellipse-pr>) es completamente diferente. Sigue sin ser perfecto, pero parece funcionar mejor.

==== 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#cite(<touch-driver>) que he escrito para ello desde 0.

Este hardware, al igual que la pantalla, también tiene una particularidad, y es que la función 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 para enviar un byte vacío y recibir la respuesta dejando una configuración conveniente.

==== 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#cite(<templarian-mdi>) 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#cite(<qgf>) se pueden encontrar en este repositorio#cite(<mdi-qgf>)

==== Añadir nuestro código

Para poder usar nuestros nuevos archivos, no es suficiente con añadir un 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. También es necesario, no tengo muy claro por qué, añadir los archivos 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 #cite(<rules-mk>)

==== 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 controlemos 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:

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 de forma que no intentemos usar la pantalla táctil o enviar un mensaje XAP si estas características no se han activado.

== 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#cite(<tauri>) (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.

==== Instalación

Para instalar Rust podemos usar pacman , 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

La comunicación con el teclado se realiza usando la librería hidapi#cite(<hidapi>), a la que accedemos desde Rust gracias al wrapper#cite(<hidapi-rs>) que implementa una "pasarela" a la librería en C. Para instalarla tan solo necesitamos añadirla al archivo de nuestro proyecto

Primero instalamos NodeJS con , después clonamos el repositorio#cite(<qmk_xap>), usado de base (y en el que he colaborado) para desarrollar el software. Para instalar las dependencias de JavaScript ejecutamos . Ahora ya podemos correr para lanzar nuestro programa.

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

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`

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

 impl XAPBroadcast for ReceivedUserBroadcast {}
    ```,
    caption: [Definición de mensaje de usuario]
)
```

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

```
#snippet(
    ```rust
 pub(crate) enum XAPEvent {
 HandleUserBroadcast {
 broadcast: BroadcastRaw,
 id: Uuid,
 },
 // -- Otros eventos recortados --
 }
    ```,
    caption: [Definición de nuevo evento]
)
```

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

```
#snippet(
    ```rust
 match broadcast.broadcast_type() {
 BroadcastType::User => {
 event_channel
 .send(XAPEvent::ReceivedUserBroadcast { id, broadcast })
 .expect("failed to send user broadcast event!");
 }
 // -- Otros tipos recortados --
 }
    ```,
    caption: [Emitir evento al recibir mensaje]
)
```

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

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

```

 }
 },
 caption: [Manejo del evento]
)

```

Toda la lógica de usuario (personalizable), se puede encontrar en el archivo `#cite(<user-rs>)`

#### **===== 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

```

#snippet(
    ```rust
    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"));
    ```,
 caption: [Creación del _systray_]
)

```

```

#snippet(
    ```rust
    .system_tray(system_tray)
    .on_system_tray_event(move |app, event|
        match event {
            // Al usar click izquierdo
            SystemTrayEvent::MenuItemClick { id, .. } => {
                // Segun boton 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
        }
    ```,
 caption: [Añadir _systray_ al programa, con su logica]
)

```

```

#snippet(
    ```rust
    .on_window_event(
        |event| match event.event() {
            tauri::WindowEvent::CloseRequested { api, .. } => {
                event.window().hide().unwrap();
                api.prevent_close()
            },
            _ => {} // Otros eventos, no hacemos nada
        }
    ```,
 caption: [Interceptar señal de cierre de ventana]
)

```

#### **===== 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

```

#snippet(
    ```Diff
    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);
    }
}

```,
caption: [Hooks de usuario]
)

===== 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 arrancar la aplicación ejecutamos:

#snippet(
```rust
pub(crate) fn on_init() {
    match std::process::Command::new("sh")
        .arg("-c")
        .arg(r#"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}")
    }
}
```,
caption: [Hook de inicio]
)

== Lineas futuras

== 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++.

== Preparar el compilador
Tras clonar el source de MicroPython , hacemos para compilar el compilador cruzado de MicroPython que nos permitirá convertir el código fuente para ser ejecutado en diferentes arquitecturas.

== 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

#cli(
```bash
$ cd ports/unix
$ make submodules
$ make
```,
caption: [Compilar MicroPython para Linux]
)

#cli(
```bash
$ 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
>>>
```,

```

```

 caption: [Ejecutar MicroPython en Linux]
)

== Compilar para RP2040
Primero instalamos un compilador necesario para la arquitectura del procesador, en mi caso (Arch Linux), el comando es y después añadimos la configuración necesaria para reportar y usar un endpoint HID, siguiendo (y adaptando) #cite(<tusb-rp2>)

Definimos en C el módulo #cite(<mod-usb-hid>);, será 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

Para poder compilar la versión de RP2040 en Arch he necesitado instalar el paquete

Por último, compilamos con
#cli(
    ````bash
    $ cd ports/rp2
    $ make submodules
    $ make clean
    $ make
    ````,
 caption: [Compilar MicroPython para RP2040]
)
Y ya podremos flashear este binario en nuestra placa de desarrollo

#bibliography(
 "bibliography.yml",
 title: "Bibliografía",
)
= Anexo B. Código fuente del informe (typst).

Aquí están adjuntas las primeras líneas del código fuente con el que he generado este documento.

#text(size: 8pt)[
 #sourcerer.code(
 raw(
 read("report.typ"),
 lang: "typst"
),
 fill: rgb("#3F3F3F"),
 numbering: false,
)
]
```