
**Universidad Nacional de Río Negro**

Escuela de Producción, Tecnología y Medio Ambiente

Ingeniería Electrónica

# Plataforma de programación y prueba para PCB

## Desarrollo de maqueta y HMI

**Alumno:** Mirko Manuel Pojmaevich
**Profesores:** Gelardi Gustavo
**Materia:** Electrónica Analógica | **Código:** B5692

Fecha de entrega: 10 de noviembre de 2022

| Rev. | Fecha | Profesor | Nota |
|---|---|---|---|
| | | | |
| | | | |

# Índice

# Índice de figuras

# 1. Resumen

***Resumen***

Se presenta un sistema automático para la validación de circuitos electrónicos. Este inyecta señales en la placa siendo probada y mide las respuestas. Es capaz de medir tanto señales digitales como niveles analógicos de tensión. Para circuitos que incluyen un microcontrolador, de tipo PIC, el sistema es capaz de programar firmware, permitiendo probar la correcta funcionalidad de los módulos internos del dispositivo. El sistema clasifica los circuitos probados separando los funcionales de los defectuosos de acuerdo al conjunto de reglas que se le den. El estado de las pruebas puede ser monitoreado mediante un HMI.

**Palabras clave: Rust, Linux, NanoPi Neo3, Pogopins, Testbed**

# 2. Marco teórico

## 2.1. SBC

SCB son las siglas en ingles de Single Board Computer, refieren a una computadora completamente contenida en una única placa de circuito impreso. El ejemplo mas popular es la RaspberryPi.

## 2.2. /dev

En el sistema de directorios de Linux el directorio **/dev** es uno que contiene archivos especiales o de tipo dispositivo. En su mayoría los archivos contenidos en este directorio permiten interactuar con el hardware, tratando a los dispositivos físicos del sistema como parte del sistema de ficheros. Esto funciona asociando los canales de lectura y escritura de los archivos con los dispositivos físicos. Por ejemplo, si se escribe al archivo **/dev/dsp**, el cual representa los parlantes, se escuchará un sonido. Para el correcto funcionamiento de los dispositivos se debe respetar el protocolo mediante el cual se interactúa con estos archivos.

Los archivos de dispositivo mas relevantes para la operación de este sistema de validación de placas son los asocioados a las GPIO y los puertos de comunicación serie.

### 2.2.1. Módulos del kernel

No todos los archivos bajo **/dev** representan un dispositivo, algunos representan extensiones del kernel de Linux. Estas extensiones generalmente interactúan con uno o mas dispositivos físicos bajo un protocolo distinto, esto con el fin de encapsular, simplificar o incluso hacer posibles ciertas funcionalidades. Por ejemplo, existe una extensión de kernel para el manejo de LCDs, estos podrían ser controlados mediante escritura a los ficheros de entrada-salida o algún fichero de un módulo de comunicación serie, en ambos casos lo que se escribe en los archivos de **/dev** no es lo que se desea obtener, sino que se debe traducir el mensaje mediante el software y realizar múltiples llamados al sistema desde el entorno de usuario para la escritura. En el caso de un módulo de kernel dedicado, se puede escribir la información final que se desea obtener, la traducción y envío del mensaje ocurrirá en espacio del kernel de forma muy eficiente con una única llamada al sistema.

## 2.3. Rust

Es un lenguaje de programación de sistemas orientado a la seguridad y la eficiencia. Rust es un lenguaje compilado de alto nivel, que no utiliza un recolector de basura, como lo hace Java por ejemplo. Permite un control fino de la memoria similar al de C o C++, pero a diferencia de estos, Rust no utiliza un paradigma de alocación y liberación explícita de memoria, sino que se basa un "Borrow checker". El "Borrow checker" consiste de un conjunto de reglas que limitan como se puede programar en Rust, respetando estas reglas se obtienen un conjunto de garantías. Estas garantías incluyen un código libre de fugas de memoria, inmunidad a condiciones de carrera y validez de los datos al los que se accede. Existen situaciones en las que el "Borrow checker" es demasiado estricto y para garantizar la seguridad opta por no compilar un código válido, antes que dar por válido un programa que pudiera fallar en algún escenario. Esto puede ser un problema cuando se tiene que acceder al hardware, por ejemplo si se quiere acceder a algún registro de la plataforma que se está utilizando, el "Borrow checker" no tiene forma de garantizar que dicho registro exista, depende del programador conocer la plataforma en la que se va a ejecutar el programa, para ello existe un tipo

de bloque de código llamado “unsafe”, todo programa contenido en un bloque con esta etiqueta será compilado sin revisión del “Borrow checker”.

Para la validación de placas se requiere que el usuario programe sus pruebas personalizadas, y el mecanismo de presentación de resultados en Rust. Para ello se sugiere tener conocimiento en las siguientes áreas y “crates” del lenguaje.Derivación de rasgos (comúnmente llamados traits por su nombre en ingles), esta es una forma de dotar a una estructura de ciertos rasgos, como se explica en [1].

# 3. Materiales utilizados en la maquetación

En esta sección se describen los elementos utilizados para la realización del proyecto, no se discute si la elección fue óptima porque estos son productos de los que se disponía con antelación.

## 3.1. NanoPi Neo3

El NanoPi Neo3 es una SBC compacta y de bajo costo. Utiliza un CPU Rockchip RK3328 de cuatro núcleos, con arquitectura ARM Cortex-A53. Cuenta con 1 ó 2 GB de memoria RAM, y Gbps Ethernet.


Figura 1: NanoPi Neo3

Esta SCB interactúa con los sensores y motores coordinando el funcionamiento de la plataforma. A su vez, esta se encarga de programar los microcontroladores, inyectar las señales y leer las respuestas digitales. Como la mayoría de SCBs no contienen un ADCs incorporado, se utiliza un dispositivo externo para la medición analógica de niveles de tensión, los que luego son comunicados mediante el protocolo I2C.

Por último, este SCB actúa como host de un servidor TCP, para que múltiples usuarios puedan acceder al HMI de forma remota.

## 3.2. PIC18F45K50

Este microcontrolador es el dispositivo externo antes mencionado, que actúa como ADC y comunica los valores de tensión medidos mediante I2C. Este resulta mucho mas capaz de lo requerido para la tarea.

## 3.3. Motorreductor

Para movilizar la cinta se utiliza un motor con una reducción 1:48 y salida en doble eje TT de marca genérica similar al que se muestra en la figura 2.

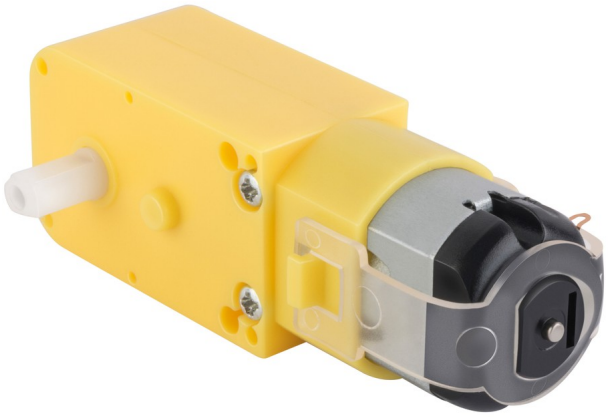


Figura 2: Motorreductor

## 3.4. Servo motor

Los servo motores utilizados son micro servos, modelo MG90S, de la marca Tower Pro, como el que se muestra en la figura 3. Este cuenta con un torque de $1,8$ kg$f$ a $4,8$ V y $2,2$ kg$f$ a 6 V.


Figura 3: Servo motor

## 3.5. Sensor óptico

Para detectar la presencia de una placa en los puntos relevantes del proceso, se utilizaron sensores ópticos. Estos consisten en un led y un fototransistor, dispuestos de manera tal que el led permita que el fototransistor sature, la detección ocurre cuando un objeto obstruye el paso de luz, causando que el fototransistor entre en corte. Su aspecto y diagrama esquemático se pueden ver en la figura 4.

Figura 4: Sensor óptico

# 4. Estructura

Para automatizar las pruebas sobre los circuitos se suele utilizar una cama de Pogopins, estas pueden estar completamente robotizadas, como la que se presenta, o pueden ser manuales, como la que se muestra en la figura 5.

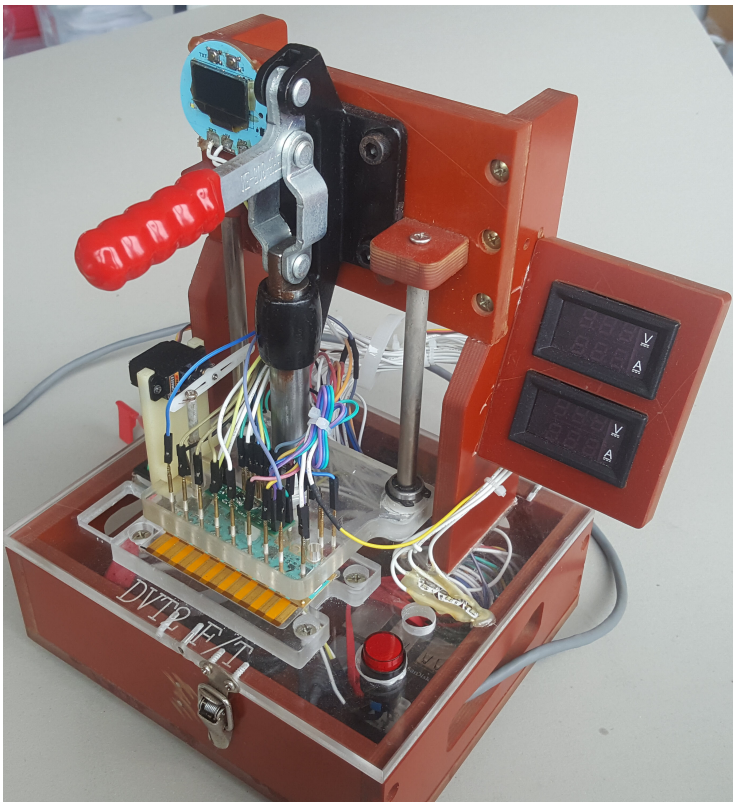


Figura 5: Ejemplo de "test jig manual"

Para la construcción de esta plataforma se decidió un diseño basado en una cinta transportadora. Esta llevaría las placas hasta que estén debajo de la cama de pines, donde un sensor óptico detectaría su presencia, deteniendo la cinta. Con la placa bajo la cama, un servo motor conectado a una leva bajaría la cama. Con los Pogopins haciendo contacto con la placa, se quemaría el firmware de prueba. Luego se inyectarían señales de acuerdo a una rutina de prueba y se medirán respuestas tanto digitales como analógicas. En base a las respuestas se clasifica la placa como valida o defectuosa. Dependiendo de la clasificación dada se posicionaría, mediante un servo motor, un selector de dirección. Por último la cinta volvería a encenderse. Para evitar que la cinta se detenga, por una placa nueva para probar, antes de que la placa ya probada llegue al selector; se utilizan dos cintas independientes, con un sensor óptico cerca del final de cada una. La primera se detiene cuando una placa llega a la posición de prueba, y la segunda cuando la última placa pasa el selector. Para su construcción, como se muestra en la figura 6, se imprimieron en 3D las piezas, que se pueden ver en la figura 7.

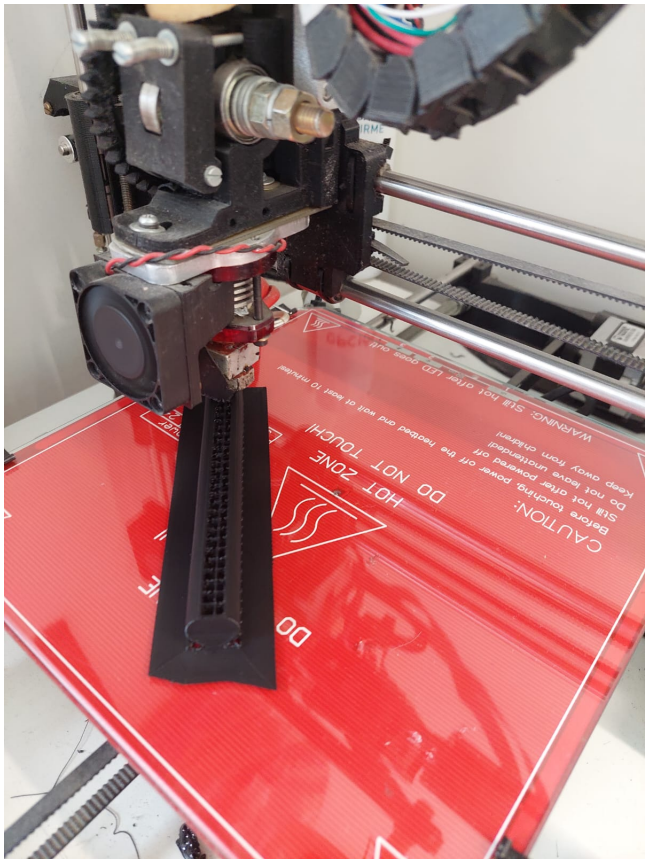



Figura 6: Impresora 3D imprimiendo una pieza

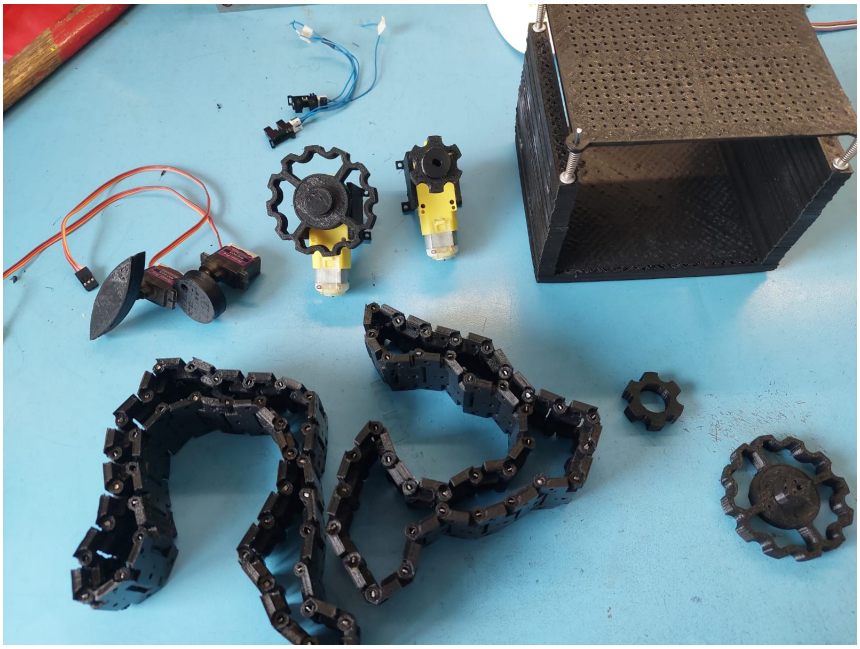


Figura 7: Piezas que componen la estructura

# 5. Pickle

Para poder realizar la programación de los PIC se utiliza el software Pickle. Este provee un mecanismo para interactuar con el microcontrolador y quemar en él un firmware. Su instalación es sencilla y puede lograrse con los comandos mostrados en el Código 1. Sin embargo, este sólo posee soporte nativo para una contada selección de SBCs populares. En el caso del NanoPi NEO3, el programa depende de un módulo externo del kernel de Linux llamado GPIO BIT BANG. Compilar módulos del kernel requiere de la instalación de los headers del kernel, los cuales están asociados al kernel que se tiene instalado, el que a su vez depende de la arquitectura en la que se trabaje. Los paquetes de headers en los repositorios de Linux suelen seguir un consenso en sus nombres, bajo la siguiente estructura, *linux-headers-$(uname -r)*. En el caso del NanoPi NEO3, este consenso se rompe siendo el nombre del paquete, *linux-headers-current-rockchip64*. Una vez instalados los headers del kernel, uno debe asegurarse de que exista un enlace simbólico, del directorio **/lib/modules/$(uname -r)/build** al directorio **/usr/src/**. Una vez que todas las herramientas para la extensión del kernel estén en su lugar, se puede instalar el módulo GPIO BIT BANG mediante los comandos mostrados en el Código 2 (el comando *hg* requiere de la instalación de mercurial).

```
1  cd /tmp
2  wget http://wiki.kewl.org/downloads/pickle-4.20.tgz
3  tar zxf pickle-4.20.tgz
4  cd pickle-4.20/
5  make
6  sudo make install
```

Código 1: Instalación Pickle

```
1  hg clone http://hg.kewl.org/pub/gpio-bb
2  cd gpio-bb
3  make
4  sudo make install
```

Código 2: Instalación GPIO BIT BANG

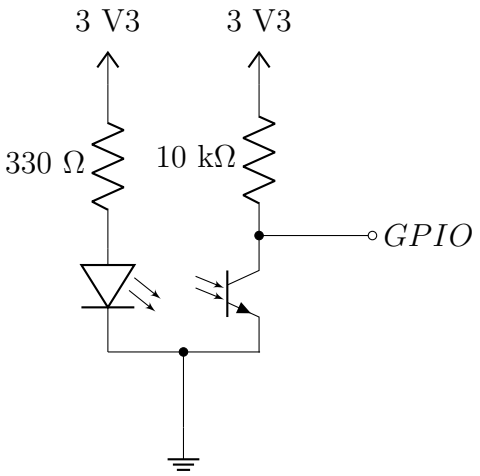
Pickle se configura por medio de un archivo llamado **.pickle** donde se especifica que mecanismo se va a utilizar para programación, así como a que pines está conectada cada línea de programación del ICSP. En el código 3 se puede ver un ejemplo de una configuración para GPIO BIT BANG. Los números asignados a cada linea indican el número de pin para Linux, el cuál varía en cada dispositivo pero suele estar bien documentado. En el caso del NanoPi Neo3, los gpio estan distribuidos en 5 grupos, estos figuran bajo **/dev**, como **gpiochipo0**, **gpiochipo1**, **gpiochipo2**, **gpiochipo3** y **gpiochipo4**. Dento de cada grupo hay 32 pines, enumerados del 0 al 31. El número de pin de Linux es único, y se calcula como $(N° \textit{ de gpiochip}) \cdot 32 + (N° \textit{ interno en el chip})$. Por ejemplo, el pin número cuatro, del gpiochip2, lleva el número $2 \cdot 32 + 4 = 68$ para Linux.

```
1 DEVICE=/dev/gpio-bb
2 IFACE=/dev/gpio-bb
3 SLEEP=1
4 BUSY=0
5 BITRULES=0x1000
6 VPP=103
7 PGM=101
8 PGC=102
9 PGD=100
```

Código 3: Ejemplo de configuración de Pickle para GPIO BIT BANG

# 6. Cinta

Para controlar la cinta se implementó un programa en Rust, el cuál funciona prendiendo y apagando un motor en función de eventos, como flancos ascendentes y descendentes en el pin conectado a un sensor óptico. La conexión con el sensor se muestra en el circuito 1. El principio de funcionamiento para las primera cinta es muy simple, prender el motor y mantenerlo así hasta ver un flanco ascendente en el pin del sensor, luego dar la orden de iniciar las pruebas, una vez terminadas las pruebas repetir el proceso. Este concepto tiene el problema de que la velocidad de transición del sensor óptico es muy lenta. Lo que causa que se detecten muchos flancos cada vez que una placa llega. La programación utilizada es orientada a eventos, no a interrupciones, por lo que cada evento de flanco es agregado a una cola FIFO y leída cuando se consulta. Esto causa que si una placa al llegar genera diez eventos, la rutina de parar la cinta, bajar la cama e intentar realizar las pruebas, se ejecutará diez veces a pesar de que hubiera una única placa. Esto se solucionó midiendo el sensor en cada evento, e ignorando los eventos donde no había placa, esta solución se muestra en el código 4, aquí el testeo de la placa se representó como una espera de cinco segundos. La solución fue efectiva cuando se estaba utilizando un led para simular el motor, una vez puesto el motor, como se muestra en el circuito 2, que se planeaba utilizar, este inyectó pulsos inesperados, que fueron erróneamente detectados como eventos, y estos podían ocurrir aún con una placa en el sensor, lo que causa que el evento no se descarte. Este nuevo problema se solucionó con un capacitor paralelo al motor. Por último, para poder integrar este principio en el resto del proceso es necesario enviar un mensaje al segundo tramo de cinta dando aviso de que una placa acaba de ser probada.

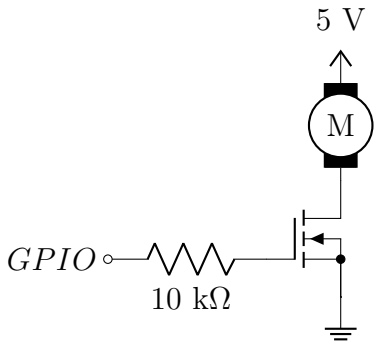



Circuito 1: Conexión del sensor óptico al NanoPi Neo3

La lógica del segundo tramo es aun más simple, esperando en un bucle el mensaje de que una placa fue probada, y ante su llegada encendiendo la cinta hasta el evento de que la placa pasó el selector. Esta implementación se muestra en el código 5.

```rust
use gpiod::{Chip, Options, EdgeDetect};
use std::time::Duration;
use std::thread::sleep;

fn main() -> std::io::Result<()> {
    let chip = Chip::new("gpiochip3")?; // open chip

    let opts = Options::output([4]) // configure lines offsets
        .values([false]) // optionally set initial values
        .consumer("my-outputs"); // optionally set consumer string

    let ipts = Options::input([6]) // configure lines offsets
        .edge(EdgeDetect::Rising)
        .consumer("my-inputs"); // optionally set consumer string

    let outputs = chip.request_lines(opts)?;
    let mut inputs = chip.request_lines(ipts)?;

    let wait = Duration::from_secs(5);

    loop{
        outputs.set_values([true])?;
        while inputs.get_values([false;1])? == [true] { }
        while inputs.get_values([false;1])? == [false] {
            let event = inputs.read_event()?;
            println!("Evento: {:?}",event);
        }
        outputs.set_values([false])?;
        sleep(wait);
    }

}
```

Código 4: Código para controlar el primer tramo de cinta




Circuito 2: Conexión del motorreductor al NanoPi Neo3

```rust
use gpiod::{Chip, Options, EdgeDetect};
use std::sync::mpsc;

fn main() -> std::io::Result<()> {
    let chip = Chip::new("gpiochip3")?; // open chip

    let opts = Options::output([4]) // configure lines offsets
        .values([false]) // optionally set initial values
        .consumer("my-outputs"); // optionally set consumer string

    let ipts = Options::input([6]) // configure lines offsets
        .edge(EdgeDetect::Falling)
        .consumer("my-inputs"); // optionally set consumer string

    let outputs = chip.request_lines(opts)?;
    let mut inputs = chip.request_lines(ipts)?;

    let (tx,rx) = mpsc::channel();

    loop{
        rx.recv(); //Recibe de forma bloqueante la salida de una placa
        outputs.set_values([true])?;
        while inputs.get_values([false;1])? == [false] { }
        while inputs.get_values([false;1])? == [true] {
            let event = inputs.read_event()?;
            println!("Evento: {:?}",event);
        }
        outputs.set_values([false])?;
    }

}
```

Código 5: Código para controlar el segundo tramo de cinta

# 7. Servos

Tanto el servo que controla la posición de la cama, como el que controla la posición del selector son controlados bajo el mismo principio. Estos se controlan con una señal PWM con periodo de 20ms y cuyo ciclo de trabajo refleja el ángulo del servo. Según la hoja de datos del motor la alimentación debe estar contenida entre 4,8 V y 6 V, y los ciclos de trabajo tener una duración contenida entre 1ms y 2ms. Como en este caso la tensión de la señal de control es distinta de la tensión de alimentación, los valores del ciclo de trabajo se midieron a mano a base de prueba y error. Ambos servos trabajan únicamente en dos posiciones, arriba-abajo para la cama y válida-inválida para el selector, se buscaron los tiempos que resultan en las posiciones deseadas y se los programó de forma rígida como una constante en el código. En un microcontrolador se acostumbra a generar las señales PWM mediante interrupciones de timer, o algún módulo de dedicado a la tarea. En el caso de Linux desde el espacio de usuario esto no resulta posible. Existe una última forma para lograr este comportamiento, que en el espacio de los microcontroladores se considera mala práctica, establecer un valor a la salida y, de forma bloqueante, esperar un tiempo determinado para restablecer el valor de la salida. Esto es considerado mala práctica porque deja bloqueada la ejecución total del código. En el caso del NanoPi Neo3, al contar este con un CPU multinúcleo y estar ejecutando los programas sobre un sistema operativo multiproceso, este problema se vuelve irrelevante. Siempre y cuando el control del PWM se ejecute en un hilo independiente, el bloqueo de la ejecución no representa ningún tipo de problema. Para lograr controlar las posiciones de los servos desde otros hilos de ejecución, se envían mensajes indicando que constante de período utilizar, para simplificar el procesamiento solo se envía un mensaje cuando se desea establecer una posición, y esta se sostiene hasta el próximo mensaje. La implementación del control PWM se muestra en el código 6.

```rust
use gpiod::{Chip, Options};
use std::time::Duration;
use std::thread::sleep;

fn main() -> std::io::Result<()> {
    let chip = Chip::new("gpiochip3")?; // open chip

    let opts = Options::output([4]) // configure lines offsets
        .values([false]) // optionally set initial values
        .consumer("my-outputs"); // optionally set consumer string

    let outputs = chip.request_lines(opts)?;

    let periodo = Duration::from_millis(20);
    let buena = Duration::from_micros(550);
    let mala = Duration::from_micros(2350);
    let resto_b = periodo-buena;
    let resto_m = periodo-mala;
    let mut cont = 0;

    loop{
        let (high,low) = if cont >= 50 {
            (buena,resto_b)
        } else {
            (mala,resto_m)
        };
        outputs.set_values([true])?;
        sleep(high);
        outputs.set_values([false])?;
        sleep(low);
        cont +=1;
        cont %= 100;
    }
}
```

Código 6: Controlar los servos mediante PWM (ejemplo del selector)

# 8. ADC

Como se mencionó anteriormente, el NanoPi Neo3 no cuenta con un ADC propio, por lo que se incorporó un PIC18F45K50, con un firmware que realiza mediciones analógicas y envía los resultados por I2C. Este firmware es uno muy simple iterando entre tres mediciones y guardándolas en un buffer interno. Ante cada lectura consecutiva por I2C, se entrega de forma cíclica un valor del buffer y se pasa al siguiente. Se puede restaurar el valor inicial del buffer escribiendo por I2C el número 42.

En cuanto al NanoPi Neo3, se realiza la comunicación utilizando un módulo de hardware dedicado al I2C. Se accede a este utilizando la interfaz **/dev/i2c-0**. En el código 7, se muestra un ejemplo donde se realizan 1000 consultas al PIC sobre cada la medición del ADC y se las promedia.

```rust
extern crate i2c_linux;
use i2c_linux::I2c;
use std::thread;
use std::time::Duration;

fn main() -> std::io::Result<()> {
    let mut i2c = I2c::from_path("/dev/i2c-0").expect("i2c");
    i2c.smbus_set_slave_address(0x08, false).expect("addr");
    let mut cont = 0;
    i2c.smbus_write_byte(42).expect("Write");//Sincronizar el pic
    let mut tensiones: [f64;3] = [0.0;3];
    for _ in 0..3000{
        let adch = i2c.smbus_read_byte().expect("ReadH") as u16;
        let adcl = i2c.smbus_read_byte().expect("ReadL") as u16;
        let adc:u16 = (adch <<8) + adcl;
        let tension = adc as f64 * 0.003225806452;
        tensiones[cont as usize] += tension;
        println!("Read I2C data{}: {}V", cont, tension);
        cont += 1;
        cont %= 3;
        thread::sleep(Duration::from_millis(1));
    }
    println!("Tension 0: {}V", tensiones[0]/1000.0);
    println!("Tension 1: {}V", tensiones[1]/1000.0);
    println!("Tension 2: {}V", tensiones[2]/1000.0);

    Ok(())
}
```

Código 7: Ejemplo de comunicación entre el NanoPi Neo3 y el PIC18F45K50 por I2C

# 9. Servidor

Todo el código de control del sistema es monitoreado dentro de un lazo infinito en un servidor TCP. La estructura del servidor se describe en la figura 8.

El lazo principal, encargado del monitoreo y manejo de las comunicaciones, se ejecuta en el hilo principal del código. Previo a entrar en dicho lazo, se lanza un hilo independiente para la gestión de la escucha del puerto. La gestión de escucha es muy simple, espera de forma bloqueante un cliente entrante. Cuando un cliente llega se crea un canal de comunicación y se envía un extremo al hilo principal y el otro extremo se envía, junto con su creación, a un nuevo hilo para el manejo del cliente. El hilo de manejo de cliente tiene que poder gestionar en simultaneo tanto los mensajes entrantes de los clientes, para enviar al hilo principal, y los mensajes del servidor para enviar a los clientes. Para reducir la carga sobre el CPU, se realizan ambas operaciones de forma bloqueante, cada una en su propio hilo. Cualquier cambio en el estado del sistema es enviado a todos los clientes conectados, y cualquier mensaje de control de un cliente llega al servidor y al sistema.

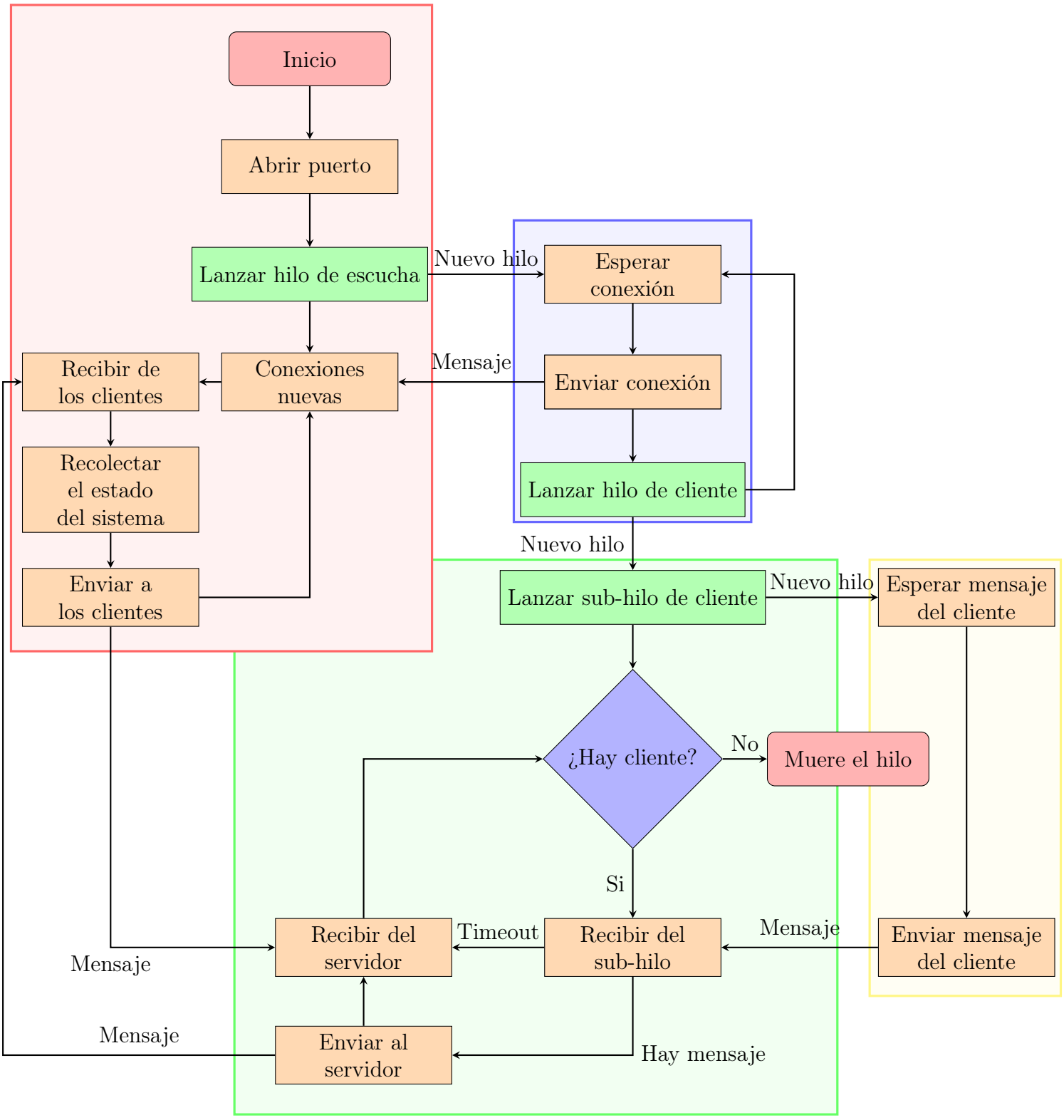



Figura 8: Diagrama de flujo del servidor (Recuadro rojo: hilo principal. Recuadro azul: hilo de escucha. Recuadro verde: hilo de gestión de cliente. Recuadro Amarillo: sub-hilo de gestión de cliente)

# 10. Cliente

El cliente sigue una lógica muy directa. Primero se conecta con el servidor, luego inicia un bucle de ejecución en tres partes. De forma no bloqueante se consulta si hay mensajes del servidor y si hay algún mensaje para enviar se lo envía. Si llegó alguna actualización del servidor se la procesa, traduciendo los datos para poder mostrarlos en pantalla. Por ultimo se genera la imagen y se pregunta si ocurrió alguna interacción por teclado, si hubo interacción se la condensa en un mensaje para el servidor y se vuelve a iniciar el bucle.

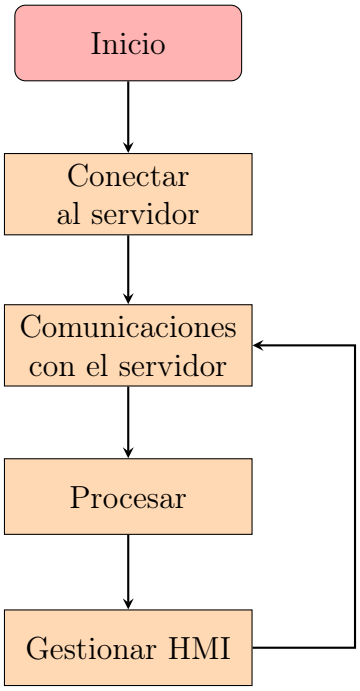



Figura 9: Diagrama de flujo del cliente

# 11. Ensamblaje del software

Se explicaron todos los módulos de código de forma independiente, pero no se mencionó como estos trabajarían en conjunto. Aprovechando lo mencionado en la sección 7, Linux al ser un sistema operativo multiproceso permite lanzar cada módulo en un hilo independiente y mantenerlos en sincronía mediante mensajes. Esto simplifica mucho la integración porque podemos saber de antemano que no tendremos problemas coordinando los módulos ya probados.

# Referencias

[1] Rust By Example. *Derive*. https://doc.rust-lang.org/rust-by-example/trait/derive.html. Leido el 22/11/2022. Jun. de 2019.