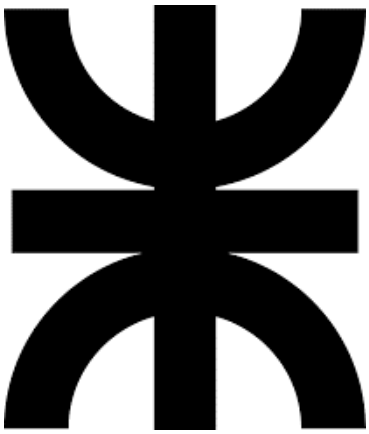
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA

INGENIERÍA ELECTRÓNICA



TÉCNICAS DIGITALES 2

Trabajo Práctico Final

SECUENCIAS DE LUCES EN RASPBERRY PI

DOCENTES	Toledo, Luis.	
	Pereira, Estefania.	
COMISIÓN	4R1	
ALUMNOS	Capdevila, Facundo.	90129
	Gomez, Enzo.	90709
	Pettiti, Matias.	89614

Córdoba, 27 de febrero de 2025

CONTENIDO

1. Introducción

El presente informe documenta el diseño, desarrollo e implementación de un sistema interactivo basado en una Raspberry Pi, utilizando las bibliotecas WiringPi y pthread. El proyecto tiene como objetivo principal implementar un programa que permita la selección y ejecución de distintas secuencias de luces LED, incorporando elementos avanzados de control, interacción y configuración. Este sistema está diseñado para ser robusto, versátil y capaz de operar en modos tanto locales como remotos.

Entre las funcionalidades principales desarrolladas se incluyen: la selección de secuencias de luces desde un menú interactivo, la gestión de acceso al menú mediante un sistema de autenticación con contraseña, y la capacidad de modificar dinámicamente la velocidad de las secuencias a través de comandos del usuario. Asimismo, se han implementado técnicas para la conservación del estado entre diferentes ejecuciones y se utiliza un conversor analógico-digital (A/D) para inicializar los valores de velocidad con base en un potenciómetro conectado al hardware.

El programa también ofrece una flexibilidad adicional mediante la configuración de modos de operación: en modo local, las secuencias de luces se ejecutan en el hardware conectado directamente a la Raspberry Pi; en modo remoto, la funcionalidad se extiende para interactuar con una PC mediante comunicación serie UART. Esta arquitectura modular permite que el sistema sea adaptable a diferentes escenarios de hardware.

A lo largo del documento se detallan los procedimientos, algoritmos y técnicas implementadas, así como las decisiones tomadas para optimizar la interacción, funcionalidad y rendimiento del sistema.

2. Objetivos

- Desarrollar un sistema de control de secuencias de luces LED interactivo.
- Implementar control de acceso seguro.
- Proveer flexibilidad en la ejecución de secuencias.
- Configurar velocidades iniciales mediante hardware.
- Incorporar modos de operación local y remoto.
- Optimizar la experiencia de usuario en términos de respuesta.
- Integrar algoritmos personalizados y tablas de datos.
- Incluir una función en lenguaje ensamblador ARM.
- Proporcionar modularidad y adaptabilidad al sistema.

3. Marco Teórico

3.1. Programación por hilos

La programación en hilos se basa en el concepto de hilos de ejecución (threads), que son unidades más pequeñas de procesamiento dentro de un proceso. Cada proceso puede tener múltiples hilos que comparten los mismos recursos (memoria, variables globales, etc.) pero pueden ejecutarse de forma independiente y simultánea.

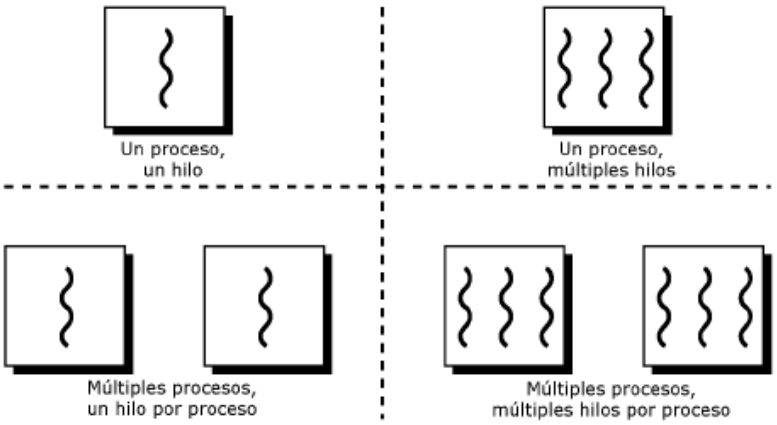


Figura 1: Procesos e hilos

3.2. Características y ventajas de usa hilos

Características:

- Ejecución concurrente: Los hilos permiten que varias tareas se ejecuten simultáneamente en el mismo programa, optimizando el uso del procesador.
- Recursos compartidos: Todos los hilos de un proceso comparten el mismo espacio de memoria, lo que facilita la comunicación, pero también requiere sincronización para evitar condiciones de carrera.
- Ligereza: Los hilos son más ligeros que los procesos, ya que comparten el mismo contexto del proceso principal.

Ventajas:

- Permite aprovechar sistemas con múltiples núcleos de CPU.
- Mejora la respuesta de aplicaciones que necesitan realizar tareas simultáneas.
- Facilita la creación de programas multitarea, como servidores concurrentes.

3.3. Librería pthread

La biblioteca pthread (POSIX Threads) es una API estándar para trabajar con hilos en sistemas compatibles con POSIX, como Linux. Facilita la creación, sincronización y gestión de hilos en aplicaciones multitarea.

Características:

- Creación de hilos: Permite dividir un programa en múltiples hilos que pueden ejecutarse en paralelo.
- Compartición de recursos: Los hilos comparten la memoria y recursos del proceso principal.
- Sincronización: Proporciona herramientas como mutexes, semáforos y variables de condición para evitar conflictos entre hilos.
- Control de hilos: Permite iniciar, detener y esperar la finalización de hilos.

Funciones principales:

- pthread_create: Crea un hilo.
- pthread_join: Espera a que un hilo termine.
- pthread_mutex_t: Sincroniza el acceso a recursos compartidos.
- pthread_exit: Termina un hilo explícitamente.

3.4. ADC 1115

El ADC1115 es un convertidor analógico a digital (ADC) de 16 bits desarrollado por Texas Instruments. Está diseñado para convertir señales analógicas en señales digitales y ofrece alta resolución y precisión. Alguna de las caracteristicas principales son:

- **Resolución de 16 bits**: Proporciona una alta precisión en la conversión de señales analógicas.
- **Interfaz I2C**: Se comunica a través del bus I2C, lo que facilita la conexión con microcontroladores como Arduino, Raspberry Pi, entre otros.
- **Bajo consumo de energía**: Es adecuado para dispositivos alimentados por batería.
- **Rango de entrada seleccionable**: El ADC1115 ofrece varias opciones de ganancia programable, como:
  - ±6.144 V
  - ±4.096 V
  - ±2.048 V
  - ±1.024 V
  - ±0.512 V
  - ±0.256 V
- **Compensador integrado**: Tiene un comparador integrado que puede generar una interrupción cuando se cruza un umbral.
- **Múltiples canales**: Dispone de cuatro entradas de señal simple o dos entradas diferenciales, lo que lo hace versátil para diferentes aplicaciones.
- **Simplicidad**: Su interfaz I2C simplifica la comunicación, y solo se necesita conectar dos cables (SCL y SDA) para su funcionamiento.

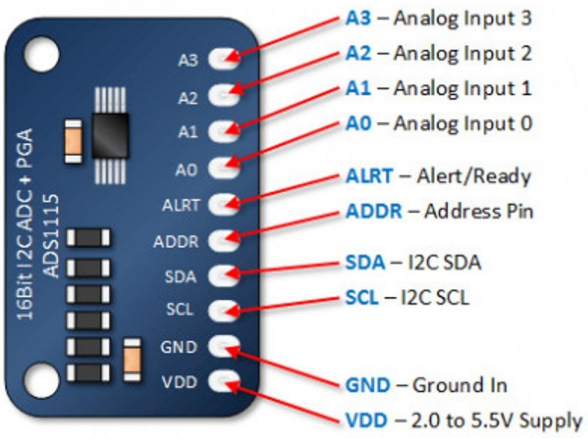


Figura 2: Pinout ADC1115

3.5. WiringPi

WiringPi Library es una biblioteca de acceso GPIO de alto rendimiento creada para las placas Raspberry Pi. Desarrollada en el lenguaje de programación C, esta biblioteca está diseñada para ofrecer un control eficiente y rápido de los pines GPIO mediante el acceso directo a los registros de hardware utilizando DMA, lo que garantiza un rendimiento óptimo.

La biblioteca es compatible con todas las placas Raspberry Pi, incluida la Raspberry Pi 5. Sin embargo, en esta última, actualmente solo se encuentra disponible la funcionalidad GCLK debido a la falta de documentación completa sobre el chip RP1. A pesar de esta limitación, WiringPi sigue siendo una solución confiable para gestionar los pines GPIO en una amplia variedad de proyectos.

Gracias a su diseño y adopción generalizada, WiringPi se ha convertido en una herramienta esencial para numerosos desarrolladores. Es utilizada tanto en tareas simples, como el parpadeo de un LED, como en sistemas más complejos de automatización. Su enfoque en el acceso directo al hardware garantiza una latencia mínima, lo que la hace ideal para aplicaciones que requieren alta velocidad y precisión en las operaciones GPIO.

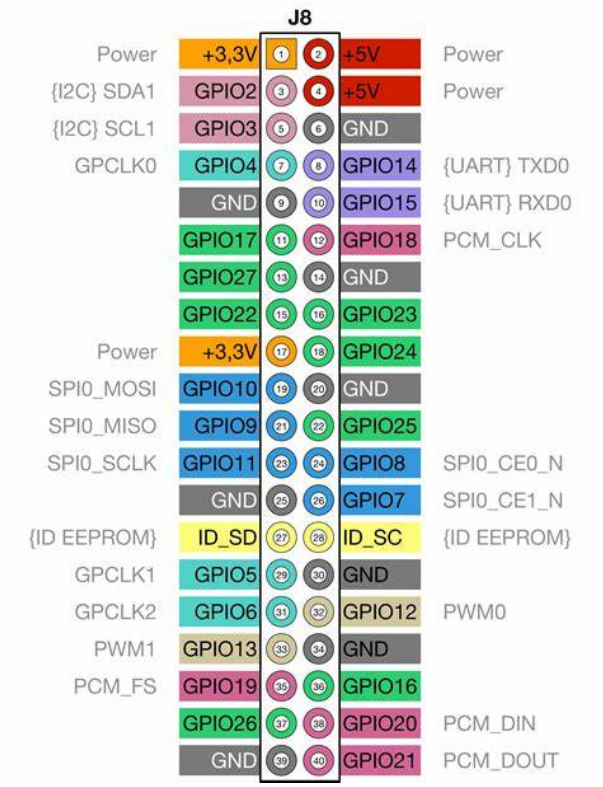


Figura 3: Pinout GPIO Raspberry Pi 5

4. Consignas

1. Realice un programa a fin de que el usuario pueda seleccionar desde un menú, una de ocho secuencias de luces posibles. Cuatro de ellas serán comunes para todos los proyectos y son: ”El auto fantástico”, ”El choque”, ”La apilada” y ”La carrera”. Las otras cuatros serán propias de cada grupo y se deberán implementar dos de ellas con algoritmo y las dos restantes por medio de la técnica de tablas de datos.

2. Implemente el control de acceso a este menú mediante password.

3. Cada vez que el usuario seleccione una secuencia el programa deberá cambiar la pantalla para indicar cual secuencia está ejecutándose y como hacer para salir de la misma. Al optar por abandonar la actual, el programa deberá regresar al menú principal inmediatamente sin completar la secuencia que se está desarrollando y apagando todas las luces.

4. Permita la posibilidad de controlar la velocidad de cada secuencia. Presionando la flecha hacia arriba se incrementará la velocidad y presionando la flecha hacia abajo se reducirá. Introduzca el sensado de las teclas oprimidas en el lugar apropiado de su programa a fin de percibir la reacción del sistema en forma inmediata, independiente de la velocidad actual. La velocidad ajustada en cada secuencia deberá conservarse entre llamadas a diferentes secuencias.

5. El valor inicial correspondiente a la velocidad de las secuencias deberá ingresarse mediante la lectura del estado de los potenciómetros que están conectados a las entradas analógicas del conversor A/D.

6. Generar una opción en el programa que permita establecer dos modos de trabajo: local y remoto. En modo local las secuencias de luces se ejecutarán en los leds que se encuentran en el hardware adicionado a la placa Raspberry donde se ejecuta el programa. Existen dos opciones para el modo remoto dependiendo de la disponibilidad de hardware de cada grupo, hacer una de las dos opciones siguientes:

   - En modo remoto las secuencias se ejecutarán sobre el hardware adicional colocado en otra Raspberry y conectada a la que ejecuta el programa mediante un cable serie RS-232. Se podrá usar el mismo programa para implementar esta opción en las dos Raspberry o realizar uno principal y otro secundario.
   - En modo remoto la selección desde el menú de la secuencia de luces a ejecutarse y control de velocidad se harán desde una PC que se encontrará conectada mediante un cable serie RS-232 a la Raspberry.

7. Como opción genere una sección destinada a establecer las velocidades iniciales de las secuencias realizando el ajuste de los potenciómetros.

8. Escriba una función externa en código ensamblador de ARM la cual será llamada desde el programa principal. Esta función deberá realizar algún proceso simple dentro del programa (switch case, loop, etc). El estudiante decidirá el proceso a resolver por la función y el lugar donde se usará.

4.1. Explicacion de secuencias

- Auto fantástico: una luz que se desplaza de izquierda a derecha y de derecha a izquierda.
- El choque: una luz desde la izquierda y otra desde la derecha comienzan su recorrido en forma opuesta y por lo tanto al medio se cruzan y continúan hasta los extremos opuestos. En la visualización de la misma parece que las luces chocan y se repelen.
- La apilada: Una luz arranca de izquierda a derecha y cuando alcanza el extremo derecho, parpadea y se queda encendida en la última posición; a partir de allí una nueva luz comienza su recorrido desde la izquierda y se desplaza hacia la derecha hasta llegar a la posición anterior a la que está fija, parpadea y también se queda quieta como la anterior. Ahora son dos las luces quietas y se repite el mismo proceso para una nueva luz arrancando por la izquierda y llegando a la posición anterior a las dos quietas. La secuencia termina cuando los ocho lugares han sido ocupados de la forma descripta.
- La carrera: Una luz arranca por la izquierda a una determinada velocidad, y cuando va por la mitad del recorrido arranca una nueva luz por la izquierda pero al doble de velocidad lo que produce que arriben al extremo derecho al mismo tiempo.

5. Principio de funcionamiento

El programa se basa en dos principios, programación con hilos y la utilización de descriptores de archivos (file descriptors). La programación multi-hilos permite la ejecución simultanea tanto de las secuencias como la lectura de caracteres para salir como subir y bajar la velocidad. Por otro lado, se utilizaron tres descriptores de archivos, uno actual, uno local y uno remoto, es decir, la consola local y la consola remota estarán habilitadas en todo momento, siendo configuradas de manera correspondiente al modo de ejecución seleccionado (Local o remoto), mientras que el descriptor de archivo actual es sobre el cual se basa la ejecución del programa y determina en que consola se hará la impresión en pantalla y la adquisición de datos.

6. Explicación de código

6.1. Libreria inc.h

Esta librería incluye las referencias a las librerías necesarias para el correcto funcionamiento del programa, como pthread para la programación de hilos, wiringPi para el control de los pines GPIO de la Raspberry Pi, y ads1115 para la lectura de señales analógicas. También define los prototipos de las funciones utilizadas a lo largo del código, asegurando que puedan ser llamadas correctamente en otras partes del programa. Además, establece constantes y etiquetas (como valores de velocidad, cantidad de secuencias y configuraciones de contraseñas) que facilitan la gestión de parámetros y la configuración del sistema.

```
//incluir librerias
#include <pthread.h>
#include <stdio.h>
#include <termios.h>
#include <wiringPi.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <wiringPiI2C.h>
#include <ads1115.h>
#include <fcntl.h>
#include <wiringSerial.h>
#include <unistd.h>

// Constantes simbolicas

#ifndef CRTSCTS
#define CRTSCTS 020000000000
#endif

#define FD_STDIN 0 // Entrada estandar

// Manejo del ADC
#define BASE 120
#define ADC_ADDR 0x48

// Manejo de la velocidad
#define VEL_INICIAL 100
```

```c
#define VEL_MAX_INICIAL 1000
#define VEL_MAX 3000
#define VEL_MIN 0

#define BAUDRATE B115200 /* Definicion de cte de velocidad de
    comnicacion serial*/

// Manejo de modos de ejecucion
#define LOC_TTY 0
#define REM_TTY 1
#define O_L_TTY 2
#define O_R_TTY 3


// Manejo de secuencias
#define NUM_SEQS 8
#define CNUM_SEQS sprintf(CNUM_SEQS, "%d", NUM_SEQS);
#define NUM_FUNCS NUM_SEQS+1
#define MAX_LEN 6

// Manejo de la contrasena
#define PWD "12345"
#define MAX_INTENTOS 2

// Teclas  de flecha
#define SPEED_UP_CHAR 'B'
#define SPEED_DOWN_CHAR 'A'

// Estructura para manejo de hilos
typedef struct
{
    int id;
    char* name;
    void (*func)();
    int vel;
    struct termios* tty;
    int fd;

} hilo_arg_t;

extern char* funcNames[NUM_SEQS];

// Configuraciones de terminal
void setNCanon(struct termios*, int);
void resetCanon(struct termios*, int);


// Configuraciones de Modos
int initRemote(struct termios*);
int initLocal(struct termios*);

// Configuraciones de ADC
void adcInit(void);

// Manejo de leds y wiringpi
void initWiringPi(int);
void off(int);
void encenderLeds(char bits, int);
```

```
86
87 // Cerrar puerto serie
88 void closeFD(int, int , struct termios*);
89
90 // Limpiar pantalla
91 void cls(int);
92
93 // Retardo
94 void nbd(int ms);
95
96 // Funciones de teclado
97 void* keyInput(void*);
98 void readChar(char*,int);
99
100 //secuencias y control de velocidad
101 void seq1(hilo_arg_t*);
102 void seq2(hilo_arg_t*);
103 void seq3(hilo_arg_t*);
104 void seq4(hilo_arg_t*);
105 void seq5(hilo_arg_t*);
106 void seq6(hilo_arg_t*);
107 void seq7(hilo_arg_t*);
108 void seq8(hilo_arg_t*);
109 void* pageVelocidadInicial(void*);
110 int setVelocidadInicial(void);
111
112 // Menues
113 void setMode(int*, struct termios*, int, int);
114 void printSeqMenu(hilo_arg_t*, int);
115
116 // Funciones auxiliares
117 int checkPwd(int); // Funcion para verificar la contrasena
118 int checkStop(pthread_mutex_t*, int*);// Funcion para verificar si se
    ha pulsado la tecla 'q'
119 void execFunc(pthread_t*, hilo_arg_t*, int);
120 void* hilo_func(void*);
121 extern int velMs(int cuentas);// Funcion para calcular la velocidad (
    ARM)
122
123 //variables y banderas
124 extern int stop;
125 extern pthread_mutex_t mut;
```

6.2. Funciones de sistema

6.2.1. setNCanon

Esta función tiene la finalidad de anular el modo canónico y el eco de la terminal correspondiente al descriptor de archivo `fd`. A su vez actualiza los valores nuevos de la configuración actual.

```
void setNCanon(struct termios *tty, int fd)
{
    if (fd == FD_STDIN)
    {
        tty[LOC_TTY].c_lflag = 0;              // anula entrada canonica y eco
        tcsetattr(fd, TCSANOW, &tty[LOC_TTY]); // actualiza con los valores nuevos de la config. TCSANOW = activar la modificacion inmediatamente
    }
    else
    {
        tty[REM_TTY].c_lflag = 0;              // anula entrada canonica y eco
        tcsetattr(fd, TCSANOW, &tty[REM_TTY]); // actualiza con los valores nuevos de la config. TCSANOW = activar la modificacion inmediatamente
    }
}
```

6.2.2. resetCanon

Esta función se encarga de resetear la configuración de la terminal correspondiente al descriptor de archivo `fd`, restaurando el modo canónico y el eco.

```
void resetCanon(struct termios *tty, int fd)
{
    if (fd == FD_STDIN)
    {
        tty[LOC_TTY].c_lflag |= (ICANON | ECHO | ECHOE); // anula entrada canonica y eco
        tcsetattr(fd, TCSANOW, &tty[LOC_TTY]);           // actualiza con los valores nuevos de la config. TCSANOW = activar la modificacion inmediatamente
    }
    else
    {
        tty[REM_TTY].c_lflag |= (ICANON | ECHO | ECHOE); // anula entrada canonica y eco
        tcsetattr(fd, TCSANOW, &tty[REM_TTY]);           // actualiza con los valores nuevos de la config. TCSANOW = activar la modificacion inmediatamente
    }
}
```

6.2.3. cls

Esta función imprime en la terminal correspondiente al descriptor de archivo `fd`, la cadena ANSI "\033[2J\033[H"la cual limpia la pantalla y regresa el cursor al inicio.

```
void cls(int fd)
{
    dprintf(fd, "\033[2J\033[H"); /* Limpiar pantalla y mover cursor al
    inicio */
}
```

6.2.4. nbd

La función `nbd(int ms)` implementa un retardo no bloqueante en milisegundos utilizando la función `nanosleep`, que permite especificar tiempos de espera en segundos y nanosegundos. La duración del retardo se define mediante el parámetro `ms`, que se descompone en segundos (`tv_sec`) y nanosegundos (`tv_nsec`) usando la estructura `timespec`.

El principal objetivo de la función es manejar de manera robusta las interrupciones causadas por señales del sistema. Si durante la ejecución de `nanosleep` ocurre una interrupción, esta devuelve un valor negativo y actualiza la estructura `ts` con el tiempo restante por esperar. En ese caso, la función entra en un ciclo que reintenta el retardo para completar el tiempo especificado originalmente. Sin embargo, antes de reintentar, verifica mediante una función externa, `checkStop`, si se ha solicitado detener el retardo. Esta verificación se realiza usando un mutex para garantizar la sincronización con otras partes del programa que puedan modificar la condición de parada.

```
void nbd(int ms)
{
    struct timespec ts;
    ts.tv_sec = ms / 1000;
    ts.tv_nsec = (ms % 1000) * 1000000;

    while (nanosleep(&ts, &ts) && errno == EINTR)
    {
        // Si es interrumpido por una senal, reintenta el retardo
    restante
        if (checkStop(&mut, &stop))
            break;
        else
            continue;
    }
}
```

6.2.5. keyInput

Esta función detecta si se presiono alguna de las siguientes teclas:

- ”q”: Si se presiona esta tecla la función cambia el estado de la bandera y se sale de la secuencia.
- ”arrow_up”: Al presionar dicha flecha se incrementa la velocidad de la secuencia.

- ”arrow_down”: Al presionar dicha flecha la velocidad disminuye.

Estas teclas pueden ser presionadas de manera local o remota, y la función distingue entre ambos casos al leer la tecla. Sin embargo, la función actúa de la misma manera al detectar la tecla, independientemente del origen.

```
void *keyInput(void *args)
{
    hilo_arg_t *seq = (hilo_arg_t *)args;
    char c;

    while (1)
    {
        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la tecla 'q'
            break;

        read(seq->fd, &c, 1); // Lee un car cter

        switch (c)
        {
        case 'q':

            if (checkStop(&mut, &stop))
                break;

            pthread_mutex_lock(&mut);
            stop = 1; // Actualiza la bandera para detener la secuencia
            pthread_mutex_unlock(&mut);

            break;
        case '\x1b':

            read(seq->fd, &c, 1);
            read(seq->fd, &c, 1);

            switch (c)
            {
            case SPEED_UP_CHAR: // Verifica si se ha pulsado la flecha hacia arriba

                if (checkStop(&mut, &stop))
                    break;

                pthread_mutex_lock(&mut);
                seq->vel++; //incrmenta la velocidad
                pthread_mutex_unlock(&mut);
                break;

            case SPEED_DOWN_CHAR:// Verifica si se ha pulsado la flecha hacia abajo

                if (checkStop(&mut, &stop))
                    break;

                pthread_mutex_lock(&mut);
                seq->vel--; // decrementa la velocidad
                if(seq->vel <= 1)
```

```
50                      seq->vel = 1;
51                  pthread_mutex_unlock(&mut);
52                  break;
53
54              default:
55                  break;
56              }
57              break;
58          default:
59              if (checkStop(&mut, &stop))
60                  break;
61              break;
62          }
63      }
64      return NULL;
65  }
```

6.2.6. checkPWD

Esta función implementa un mecanismo seguro para la validación de contraseñas, integrando interacción simultánea con consolas local y remota. Permite al usuario ingresar una clave y verificarla contra una contraseña almacenada de referencia. Utiliza un número limitado de intentos definido por la constante `MAX_INTENTOS` y establece restricciones de longitud máxima para la clave mediante la constante `MAX_LEN`.

El proceso inicia configurando los terminales (local y remoto) en modo no canónico mediante la función `setNCanon`, lo que permite capturar caracteres sin necesidad de presionar Enter. La clave ingresada se almacena en el arreglo `clave` y se compara con la contraseña correcta `contrasena`, definida como una constante.

Durante cada intento, se solicita al usuario que ingrese su contraseña. Los caracteres se capturan uno a uno con `read()` desde la consola correspondiente (identificada por su descriptor de archivo `fd`) y se procesan en tiempo real. Si el usuario presiona Enter (`\n`), se finaliza la captura y se agrega un carácter nulo (`\0`) al final de la cadena. Si presiona la tecla de retroceso (`127` o `\b`), el último carácter ingresado se elimina, actualizando el índice y la visualización de ambas consolas mediante movimientos del cursor y espacios en blanco.

Cada carácter válido ingresado, si no excede la longitud máxima permitida, se almacena en el arreglo y se imprime un asterisco (`*`) en ambas consolas para proteger visualmente la contraseña. Una vez finalizada la captura, la clave ingresada se compara con la contraseña correcta usando `strcmp`. Si coinciden, se imprime un mensaje de bienvenida en ambas consolas, y el bucle de intentos se interrumpe. En caso contrario, la función `cls` limpia la pantalla de ambas consolas, y se muestra un mensaje de error.

El usuario dispone de un número limitado de intentos. Si este límite se supera, se imprime un mensaje indicando que se alcanzó el máximo permitido, se restaura el modo canónico con la función `resetCanon`, y el programa finaliza con un código de error.

La función garantiza que las consolas vuelvan a su configuración original utilizando `resetCanon` antes de finalizar. Este diseño asegura un manejo seguro de contraseñas y una interacción coherente entre consolas local y remota.

```c
int checkPwd(int fd)
{
    char clave[MAX_LEN + 1];            // Array para almacenar la clave ingresada
    char contrasena[MAX_LEN + 1] = PWD; // Contrasena correcta
    char c;
    int i, resultado;

    cls(fd);
    tcflush(fd, TCIFLUSH);
    for (i = 0; i < MAX_INTENTOS; i++)
    {

        dprintf(fd, "Ingrese su contrase a: ");
        int pos = 0; // Indice para la clave

        while (1)
        {
            readChar(&c, fd); // Leer un caracter

            if (c == '\n')
            {                       // Detectar Enter
                clave[pos] = '\0'; // Terminar la cadena
                break;             // Salir del bucle si se presiona Enter
            }
            else if (c == 127 || c == '\b')
            { // Detectar Backspace
                if (pos > 0)
                {
                    pos--;                 // Disminuir indice
                    dprintf(fd, "\b \b"); // Mover el cursor atras y borrar el asterisco
                }
            }
            else
            {
                if (pos < MAX_LEN)
                {                       // Asegurar que no se exceda la longitud maxima
                    clave[pos++] = c; // Agregar el caracter a la clave
                    dprintf(fd, "*"); // Imprimir un asterisco en la pantalla
                }
            }
        }
        dprintf(fd, "\n");

        // Comparar la clave ingresada con la contrasena
        resultado = strcmp(clave, contrasena);
        if (resultado == 0)
        {
            dprintf(fd, "Bienvenido\n");
            break; // Salir del bucle de intentos
        }
        else
        {
```

```
53              cls(fd);
54              dprintf(fd, "Contrase a incorrecta\n");
55          }
56
57          if (i == MAX_INTENTOS - 1)
58          {
59              dprintf(fd, "Ha superado el l mite de intentos\n");
60              return -1;
61          }
62      }
63      return 0;
64  }
```

6.3. Funciones para el manejo de hilos

6.3.1. hilo_func

La función hilo_func es el punto de entrada para un hilo creado con pthread. Recibe un argumento genérico que se convierte en un puntero a una estructura llamada hilo_arg_t. Esta estructura contiene los datos necesarios para el hilo y una función (func) que define lo que debe hacer.

Dentro de hilo_func, se llama a la función que apunta `args->func`, pasándole como argumento la misma estructura. Esto permite que cada hilo ejecute tareas específicas según los datos y la función definidos en la estructura. Al final, el hilo devuelve NULL indicando que terminó su trabajo. Es una forma flexible de manejar múltiples hilos con comportamientos personalizados.

```
void *hilo_func(void *arg)
{
    hilo_arg_t *args = (hilo_arg_t *)arg;
    args->func(args);

    return NULL;
}
```

6.3.2. execFunc

La función execFunc crea y gestiona dos hilos usando la biblioteca pthread. Recibe un arreglo de hilos (hilos), un arreglo de estructuras de argumentos (args) y un identificador (id) para seleccionar los argumentos correspondientes.

Se crean dos hilos: uno que ejecuta la función keyInput y otro que ejecuta hilo_func, ambos con los argumentos de la posición args[id]. Si la creación de alguno de los hilos falla, la función termina inmediatamente. Luego, espera a que los dos hilos terminen su ejecución usando pthread_join.

```
void execFunc(pthread_t *hilos, hilo_arg_t *args, int id)
{
    if (pthread_create(&hilos[0], NULL, keyInput, &args[id]) != 0)
        return;
    if (pthread_create(&hilos[1], NULL, hilo_func, &args[id]) != 0)
        return;
    pthread_join(hilos[0], NULL);
    pthread_join(hilos[1], NULL);
}
```

6.3.3. checkStop

Esta función se encarga de modificar la bandera Stop, la cual se utiliza cuando se presiona la tecla ”q” para salir.

```
int checkStop(pthread_mutex_t *mut, int *stop_flag)
{
    int stop = 0;

    pthread_mutex_lock(mut);
    if (*stop_flag == 1)
    {
        stop = 1;// Actualiza la bandera para detener la secuencia
    }

    pthread_mutex_unlock(mut);
    return stop;// Devuelve la bandera
}
```

6.4. Menúes

Consta de dos funciones que imprimen en pantalla dos menúes. Uno permite distinguir entre el modo remoto o local y el segundo es para distinguir entre las secuencias y el ajuste de velocidad.

6.4.1. printSeqMenu

La función "printSeqMenu" genera un menú interactivo en la consola que permite al usuario seleccionar una opción relacionada con secuencias almacenadas en un arreglo de estructuras seqs del tipo hilo_arg_t.

Primero, limpia la pantalla con cls(). Luego, imprime un encabezado decorativo seguido de las opciones del menú. Estas incluyen: salir (opción 0), cambiar modo(opción q), una lista de secuencias numeradas (extraídas del campo name de cada elemento en seqs), y una opción adicional para configurar la velocidad. Finalmente, muestra un mensaje para que el usuario seleccione una opción.

```
void printSeqMenu(hilo_arg_t *seqs, int fd)
{
    cls(fd);
    dprintf(fd, "---------------------------------\n");
    dprintf(fd, "               Menu               \n");
    dprintf(fd, "---------------------------------\n");
    for (int i = 0; i < NUM_SEQS; i++)
    {
        dprintf(fd, "%d-------->%s\n", i + 1, seqs[i].name);
    }
    dprintf(fd, "---------------------------------\n");
    dprintf(fd, "%d-------->Configurar velocidad\n", NUM_SEQS + 1);
    dprintf(fd, "q-------->Cambiar modo\n");
    dprintf(fd, "0-------->Salir\n");
    dprintf(fd, "---------------------------------\n");
    dprintf(fd, "Seleccione una secuencia: ");
}
```

6.4.2. printModeMenu

Toma como argumento el descriptor de archivo actual e imprime un menu para la selección de modo de operación, local o remoto. Como también la opción para salir del programa.

```
void printModeMenu(int fd)
{
    cls(fd);
    dprintf(fd, "---------------------------------\n");
    dprintf(fd, "       Modos de ejecucion        \n");
    dprintf(fd, "---------------------------------\n");
    dprintf(fd, "1-------->Local\n");
    dprintf(fd, "2-------->Remoto\n");
    dprintf(fd, "---------------------------------\n");
    dprintf(fd, "0-------->Salir\n");
    dprintf(fd, "Seleccione un modo: ");
}
```

6.5. Fuciones para el manejo de leds

Con el fin de no trabajar directamente con el numero de los pines del GPIO en distintas partes del codigo y que las demas funciones solo sepan como usar estas funciones se realiza una abstraccion.

Tambien, el tener funciones especificas para controlar los leds hace la realización del programa mas simple.

Como se utilizara en todas estas funciones el numero de pin donde se conectaron los leds se decidió hacer un arreglo global que contenga estos. Esto permite que si se requiere utilizar otros pines solo se modifique en este lugar.

6.5.1. initWiringPi

Configura el entorno WiringPi para utilizar la numeración GPIO física de la Raspberry Pi. A continuación, declara como salidas los pines especificados en el arreglo leds mediante pinMode. Finalmente, llama a la función off, asegurándose de que todos los LEDs estén apagados y enviando un salto de línea inicial a la consola identificada por el descriptor de archivo fd.

```
void initWiringPi(int fd)
{
    wiringPiSetupGpio();
    for (int i = 0; i < 8; i++)
    {
        pinMode(leds[i], OUTPUT);
    }
    off(fd);
}
```

6.5.2. off

Apaga todos los LEDs conectados a los pines especificados en el arreglo leds. Para lograrlo, recorre este arreglo y establece en bajo (0) la salida de cada pin usando digitalWrite. Adicionalmente, envía un salto de línea (\n) a la consola identificada por el descriptor de archivo fd, permitiendo que la consola asociada registre este evento. Este descriptor de archivo puede corresponder a la consola local o remota, dependiendo de cómo se invoque la función.

```
void off(int fd)
{
    for (int i = 0; i < 8; i++)
    {
        digitalWrite(leds[i], 0);
    }
    dprintf(fd, "\n");
}
```

6.5.3. encenderLeds

La función encenderLeds controla el estado de 8 LEDs físicos y su representación en la consola asociada al descriptor de archivo fd.

Utiliza un bucle para recorrer cada bit del byte bits, determinando si debe encender ([#]) o apagar ([ ]) el LED correspondiente mediante operaciones de máscara lógica. Actualiza simultáneamente los LEDs físicos con digitalWrite y la consola con dprintf, asegurando sincronización entre ambos.

```
void encenderLeds(char bits, int fd)
{
    dprintf(fd, "\t");
    for (int i = 0; i < 8; i++)
    {
        if (bits & (1 << (7 - i)))
        {
            dprintf(fd, "[#]");
            digitalWrite(leds[i], 1);
        }
        else
        {
            dprintf(fd, "[ ]");
            digitalWrite(leds[i], 0);
        }
    }
    dprintf(fd, "\n");
}
```

6.6. Funcion en ARM

esta función lo que hace es convertir las cuentas devueltas por el ADC a milisegundos. Por lo que como argumento se pasan las cuentas y devuelve un valor entero en milisegundos.

La función hace la siguiente operación:

$$respuesta = \frac{cuentas \cdot 1001}{26367} \tag{1}$$

```
velMs:
    MOV R1, #1001          ; Carga 1001 en R1.
    MUL R0, R0, R1         ; R0 = R0 * R1.

    MOV R1, #26367         ; Carga 26367 en R1.
    UDIV R0, R0, R1        ; R0 = R0 / R1.
```

Es importante destacar que al poder utilizar las operaciones ”UDIV” y ”MUL” se reduce el código de manera significativa.

6.7. Función principal

La función principal organiza la ejecución de un sistema interactivo para controlar secuencias de luces LED en una Raspberry Pi. Configura recursos como ADC y UART para gestionar velocidades iniciales y modos de operación (local o remoto). A través de un menú interactivo, permite seleccionar secuencias, ajustar parámetros y salir, todo con validación de contraseña. Utiliza hilos para manejar tareas concurrentes como ajuste de velocidad y ejecución de secuencias sin interrumpir el flujo del programa. Finalmente, asegura un cierre ordenado liberando recursos y restaurando configuraciones, integrando control remoto, flexibilidad y respuesta en tiempo real.

```c
#include "../inc/inc.h"
int main(void)
{

    pthread_t hilos[2];
    hilo_arg_t seqs[NUM_FUNCS];
    struct termios tty[4];

    int velInicial = VEL_INICIAL;
    void (*funcs[NUM_SEQS])(hilo_arg_t *) = {seq1, seq2, seq3, seq4,
   seq5, seq6, seq7, seq8};
    int fd, lfd, rfd, opt_num;
    char opt;
    int quit = 0;

    adcInit();

    lfd = initLocal(tty);
    rfd = initRemote(tty);

    fd = lfd;
    while (quit == 0)
    {
        setMode(&fd, tty, lfd, rfd);
        if (fd == -1)
        {
            cls(lfd);
            cls(rfd);
            closeFD(lfd, rfd, tty);
            exit(1);
        }

        dprintf(fd, "%d", fd);
        initWiringPi(fd);
        for (int i = 0; i < NUM_FUNCS; i++)
        {
            seqs[i].name = funcNames[i];
            seqs[i].id = i;
            seqs[i].func = funcs[seqs[i].id];
            seqs[i].vel = velInicial;
            seqs[i].fd = fd;
        }
        setNCanon(tty, fd);
        if (checkPwd(fd) == -1)
            quit = 1;
        resetCanon(tty, fd);
```

```
46
47          while (quit == 0)
48          {
49              printSeqMenu(seqs, fd);
50              read(fd, &opt, 1);
51              tcflush(fd, TCIFLUSH);
52              opt_num = opt - '0';
53              switch ((int)opt_num)
54              {
55              case 0:
56                  cls(lfd);
57                  cls(rfd);
58                  closeFD(lfd, rfd, tty);
59                  exit(1);
60                  break;
61              case 1 ... NUM_SEQS:
62                  setNCanon(tty, fd);
63                  execFunc(hilos, seqs, opt_num - 1);
64                  resetCanon(tty, fd);
65                  break;
66              case NUM_SEQS + 1:
67                  hilo_arg_t velIArgs;
68                  velIArgs.fd = fd;
69                  setNCanon(tty, fd);
70                  if (pthread_create(&hilos[0], NULL, keyInput, &velIArgs
    ) != 0)
71                      break;
72                  if (pthread_create(&hilos[1], NULL,
    pageVelocidadInicial, &velIArgs) != 0)
73                      break;
74                  pthread_join(hilos[0], NULL);
75                  pthread_join(hilos[1], NULL);
76                  for (int i = 0; i < NUM_SEQS; i++)
77                  {
78                      seqs[i].vel = velIArgs.vel;
79                  }
80                  resetCanon(tty, fd);
81                  break;
82              case 113 - 48:
83                  quit = 1;
84              default:
85                  break;
86              }
87          }
88          quit = 0;
89      }
90      cls(lfd);
91      cls(rfd);
92      closeFD(lfd, rfd, tty);
93      return 0;
94  }
```

6.8. Secuencias

Todas las secuencias reciben como argumento una estructura la cual contiene:

- int id: Identificador de la secuencia.
- char* name: Nombre de la secuencia.
- void (*func)():función que se asigna a cada secuencia.
- int vel: Velocidad inicial de la secuencia.
- struct termios* tty: Configuración de consola y puerto serie.
- int fd: Bandera que indica en que modo estoy, si fd=0 estoy en consola local.

A través del acceso a estos datos, se realizará la impresión del nombre de la secuencia, y se asignará tanto la velocidad inicial de la secuencia como la velocidad actual. Esta última podrá variarse utilizando las teclas de flecha del teclado.

Existen dos tipos de secuencias de luces: unas implementadas mediante tablas y otras generadas por algoritmos. Tanto en las secuencias por tabla como en las generadas por algoritmo, el instante de tiempo está representado por un char de 8 bits sin signo, que corresponde al estado de los LEDs.

Dado que la función "encenderLeds" recibe un char, se tomó la decisión de trabajar los algoritmos utilizando desplazamientos y operaciones a nivel de bits. Esto facilita el desarrollo de la parte visual de la aplicación.

Algunas funciones que utilizan la mayoría de las secuencias son:

- checkStop: Maneja una bandera la cual cambia y vuelve al menú cuando se presiona la tecla "q".
- cls: Utiliza la función "clear" para limpiar la consola o el puerto serie dependiendo de fd y obtener un mejor aspecto visual.
- nbd: Esta función recibe la velocidad a la cual trabaja la secuencia, cumple la función de retardo.
- dprintf: Imprimen en consola o puerto serie dependiendo del valor del argumento int fd.
- off: Se encarga de apagar todos los leds cuando termina la secuencia.

Estas funciones ya fueron desarrolladas y es importante comprendedlas ya que, se utilizan a lo largo de todas las secuencias.

6.8.1. Auto Fantástico

Esta secuencia se esta realizando por medio de algoritmos. Posee dos for, uno ascendente que realiza el corrimiento de un bit a la derecha por iteración y el otro es un for descendente el cual realiza el corrimiento de un bit a la izquierda. Esto realiza que los leds se prendan y se apaguen desplazándose a la derecha y luego llegado al final se desplacen asía la izquierda. Estos for están dentro de un bucle infinito para repetir la secuencia.

```c
void seq1(hilo_arg_t *args)
{
    unsigned char bits = 0b10000000;
    while (1)
    {

        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la tecla 'q'
            break;

        for (int i = 0; i < 7; i++)
        {
            if (checkStop(&mut, &stop))
                break;

            cls(args->fd);// Limpia la pantalla
            pthread_mutex_lock(&mut);
            //imprime el nombre de la secuencia
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(bits >> i, args->fd); // Enciende/Apaga los LEDs
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(args->vel); // Retardo
            pthread_mutex_unlock(&mut);
        }

        for (int i = 7; i >= 1; i--)
        {
            if (checkStop(&mut, &stop))
                break;

            cls(args->fd);
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta ");
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(bits >> i, args->fd);
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(args->vel);
            pthread_mutex_unlock(&mut);
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd); // Apaga los LEDs
}
```

6.8.2. Apilada

Esta secuencia consta de dos bucles for anidados: el primero es decreciente, mientras que el segundo es creciente. Además, se tienen dos variables de tipo unsigned char:

- `unsigned char bits = 0b10000000`
- `unsigned char bits2 = 0b00000000`

A la función ”encenderLeds” se le pasa la operación OR entre las variables bits2 y el corrimiento de bits. Esto hace que los LEDs se enciendan y se apagan a medida que se desplazan hacia la derecha. Cuando el último LED se enciende, se actualiza la mascara de bits2, encendiendo el último bit correspondiente al LED más a la derecha. Al actualizarse esta variable, en la siguiente iteración se repite el proceso, pero ahora con el LED más a la derecha siempre encendido.

Este proceso está envuelto en un bucle infinito para que no se detenga hasta que se presione la tecla ”q”.

```
void seq2(hilo_arg_t *args)
{

    while (1)
    {
        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la tecla 'q'
            break;
        unsigned char bits = 0b10000000;
        unsigned char bits2 = 0b00000000;
        for (int j = 8; j >= 0; j--)
        {
            for (int i = 0; i < j; i++)
            {
                cls(args->fd);//Limpia la pantalla
                if (checkStop(&mut, &stop))
                    break;
                phtread_mutex_lock(&mut);
                //imprime el nombre de la secuencia
                dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
                dprintf(args->fd, "Presione q para volver al menu\n\n");
                encenderLeds(bits2 | bits >> i, args->fd);//Enciende/Apaga los LEDs
                dprintf(args->fd, "Tiempo: %d\n", args->vel);
                nbd(args->vel);//Retardo
                pthread_mutex_unlock(&mut);
                if (i == j - 1)
                {
                    bits2 = bits2 | bits >> i;// actualiza el valor de bits2
                }
            }
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd);// Apaga los LEDs
}
```

6.8.3. Choque

En este caso la secuencia esta realizada por medio de tabla por lo que se define la tabla con bits de cada mascara prendido y/o apagados. Luego de esto por medio de un for ascendente se le pasa a la funcion ”encenderLeds” cada uno de los elementos de la tabla para encender y apagar los leds que correspondan.

El for posee tantas iteraciones como elementos tiene la tabla.

Al igual que en todas las secuencias, esta se envuelve en un bucle infinito que frena cuando se presiona la tecla ”q”.

```
void seq3(hilo_arg_t *args)
{
    char tabla[6] = {0x81, 0x42, 0x24, 0x18, 0x24, 0x42};

    while (1)
    {
        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la
   tecla 'q'
            break;

        for (int i = 0; i < 6; i++)
        {
            cls(args->fd);//Limpia la pantalla

            if (checkStop(&mut, &stop))
                break;
            //imprime el nombre de la secuencia
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(tabla[i], args->fd);//Enciende/Apaga los LEDs
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(args->vel);//Retardo
            pthread_mutex_unlock(&mut);
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd);// Apaga los LEDs
}
```

6.8.4. Carrera

Nuevamente esta secuencia se realiza por medio de tabla por lo que se define la tabla y luego utilizando un for, en este caso de 16 iteraciones, se llama a la función ”encenderLeds” a la cual se le pasa cada elemento de la tabla y la función prende y apaga leds corresponden.

```
void seq4(hilo_arg_t *args)
{

    char tabla[16] = {0b10000000, 0b10000000, 0b01000000, 0b01000000,
                      0b00100000, 0b00100000, 0b00010000, 0b00010000,
                      0b10001000, 0b01001000, 0b00100100, 0b00010100,
                      0b00001010, 0b00000110, 0b00000011, 0b00000001};

    while (1)
    {

        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la
   tecla 'q'
            break;

        for (int i = 0; i < 16; i++)
```

```
        {
            if (checkStop(&mut, &stop))
                break;

            cls(args->fd);// Limpia la pantalla
            //imprime el nombre de la secuencia
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(tabla[i], args->fd);//Enciende/Apaga los LEDs
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(args->vel);//Retardo
            pthread_mutex_unlock(&mut);
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd);// Apaga los LEDs
}
```

6.8.5. Salto

Esta secuencia se realiza por medio de tabla por lo que la explicación es la misma que las anteriores. Sin embargo, hay una pequeña diferencia. En este caso en cada iteración se hace un doble llamado a la función ”encenderLeds”. En el primero, el argumento de la función es un elemento de la tabla con un bit en 1, encendiendo este led, mientras que en el segundo llamado el argumento es ”0” apagando todos los leds. Entre medio de estos llamados se encuentra un retardo para lograr visualizar el cambio.

```
void seq5(hilo_arg_t *args)
{
    char table[9] = {0b00000000, 0b10000000, 0b00100000, 0b00001000,
                     0b00000010, 0b00000001, 0b00000100, 0b00010000,
                     0b01000000};

    while (1)
    {
        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la
   tecla 'q'
            break;

        for (int i = 0; i < 9; i++)
        {
            cls(args->fd);// Limpia la pantalla
            if (checkStop(&mut, &stop))
                break;
            // imprime el nombre de la secuencia
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(table[i], args->fd);// Enciende/Apaga los LEDs
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(args->vel);// Retardo
            pthread_mutex_unlock(&mut);

            cls(args->fd);// Limpia la pantalla
```

```
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(table[0], args->fd);// Apaga todos los LEDs
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(10);// Retardo
            pthread_mutex_unlock(&mut);
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd);// Apaga los LEDs
}
```

6.8.6. Zig-Zag

La siguiente secuencia se realiza mediante algoritmo. Primero la variable ”bits” se inicializa como un número binario (0b10000000), equivalente a 128 en decimal. Este valor se usa como patrón inicial para encender los LEDs, y se manipulará durante la ejecución. Luego se inicializan las variables ”par” e ”impar” que seran los indices de la secuencia. Posteriormente se utiliza un for, dentro del cual codifica un if/ else que se describe a continuación:

- Si i es par: Se desplaza el patrón (bits) hacia la derecha usando par como índice y se encienden los LEDs con encenderLeds($bits \gg par$). Y el indice par se incrementa para mover el patrón en la siguiente iteración.
- Si i es impar: Se desplaza el patrón hacia la derecha usando impar como índice y se encienden los LEDs con encenderLeds($bits \gg impar$). El indice impar se decrementa para mover el patrón en la siguiente iteración.

```
void seq6(hilo_arg_t *args)
{
    unsigned char bits = 0b10000000;

    while (1)
    {
        int par = 0;
        int impar = 7;

        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la
   tecla 'q'
            break;

        for (int i = 0; i < 8; i++)
        {
            cls(args->fd);// Limpia la pantalla
            if (checkStop(&mut, &stop))
                break;

            if ((i % 2) == 0)// Verifica si es par
            {
                // imprime el nombre de la secuencia
                pthread_mutex_lock(&mut);
                dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name
   );
```

```
                dprintf(args->fd, "Presione q para volver al menu\n\n");

                encenderLeds(bits >> par, args->fd);// Enciende leds pares
                dprintf(args->fd, "Tiempo: %d\n", args->vel);
                par++;// Incrementa la variable par

            }
            else
            {
                // imprime el nombre de la secuencia
                pthread_mutex_lock(&mut);
                dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
                dprintf(args->fd, "Presione q para volver al menu\n\n");

                encenderLeds(bits >> impar, args->fd);// Enciende leds impares
                dprintf(args->fd, "Tiempo: %d\n", args->vel);
                impar--;// Decrementa la variable impar
            }
            nbd(args->vel);// Retardo
            pthread_mutex_unlock(&mut);
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd);// Apaga los LEDs
}
```

6.8.7. Vumetro

Esta secuencia se realiza mediante un algoritmo. Se utiliza un bucle for que mantiene encendido el LED más significativo mediante la función "encenderLeds(0b10000000)".

Posteriormente, mediante la función "encenderLeds(0b11111111, args->fd)", se realiza un desplazamiento de bits hacia la izquierda (i veces). Esto provoca que el patrón de LEDs encendidos cambie en cada iteración. Inicialmente, todos los LEDs están encendidos (0b11111111), y a medida que i aumenta en el for, el desplazamiento apaga progresivamente a estos.

```
void seq7(hilo_arg_t *args)
{

    while (1)
    {
        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la tecla 'q'
            break;

        for (int i = 0; i < 8; i++)
        {
            if (checkStop(&mut, &stop))
                break;

            cls(args->fd);// Limpia la pantalla
```

```
            // imprime el nombre de la secuencia
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");

            encenderLeds(0b10000000, args->fd);// Enciende el primer
    led
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            pthread_mutex_unlock(&mut);

            cls(args->fd);
            pthread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(0b11111111 << i, args->fd);// Enciende todos
    los LEDs y los apaga de atra s para adelante
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
            nbd(args->vel);// Retardo
            pthread_mutex_unlock(&mut);
        }
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    off(args->fd);// Apaga los LEDs
}
```

6.8.8. Baliza

La secuencia se realiza mediante una tabla en la que se definen los bits de cada máscara, indicando cuáles están encendidos y cuáles apagados. Posteriormente, mediante un for ascendente, se pasa a la función ”encenderLeds” cada uno de los elementos de la tabla para encender y apagar los LED correspondientes.

El for tiene tantas iteraciones como elementos contiene la tabla.

Como en todas las secuencias, esta se encuentra envuelta en un bucle infinito que se detiene cuando se presiona la tecla ”q”.

```
void seq8(hilo_arg_t *args)
{
    char table[2] = {0b11110000, 0b00001111};

    while (1)
    {

        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la
    tecla 'q'
            break;
        for (int i = 0; i < 2; i++)
        {
            if (checkStop(&mut, &stop))
                break;
            cls(args->fd);// Limpia la pantalla
            // imprime el nombre de la secuencia
            phtread_mutex_lock(&mut);
            dprintf(args->fd, "...Se ejecuta %s...\n\n", args->name);
            dprintf(args->fd, "Presione q para volver al menu\n\n");
            encenderLeds(table[i], args->fd);
            dprintf(args->fd, "Tiempo: %d\n", args->vel);
```

```
21              nbd(args->vel);// Retardo
22              phtread_mutex_unlock(&mut);
23          }
24      }
25      stop = 0;// Actualiza la bandera para detener la secuencia
26      off(args->fd);// Apaga los LEDs
27  }
```

6.9. Manejo de ADC

Para el manejo del ADC se utiliza la librería "ads1115.h" proporcionada por WiringPi.

Como se puede observar en el siguiente código, el manejo del ADC consta de dos funciones principales. La primera es "adcInit" y la segunda "setVelocidadInicial".

```
void adcInit(void)
{
    wiringPiSetup();

    if (ads1115Setup(BASE, ADC_ADDR) == -1)
    {
        printf("Error al inicializar el dispositivo I2C\n");
    }
}

int setVelocidadInicial(void)
{

    int cuentas;

    cuentas = analogRead(BASE);

    if (cuentas == -1)
    {
        printf("Error al leer el valor del ADC\n");
        return -1;
    }

    return cuentas;
}
```

- adcInit: Es la función encargada de iniciar la librería WiringPi y también de establecer la conexión con el ADC por medio de la función "ads1115Setup". A esta última función se le pasa la base (un número entero, sin especificar) y la dirección (puerto serie) que se le asigna a la Raspberry Pi.
- setVelocidadInicial: Esta función utiliza un entero para guardar las cuentas del ADC. También, utilizando la función "analogRead", la cual proviene de la librería "ads1115.h", se lee el valor del pin A0 proveniente del ADC. Este valor se guarda en la variable de cuentas y se verifica que la lectura sea correcta. Por último, la función retorna este valor.

6.9.1. Configurar velocidad inicial

La función recibe un puntero a una estructura, la cual contiene entre sus miembros una variable "vel"(velocidad), esta es la que se actualiza utilizando el potenciómetro conectado al ADC.

Se verifica si se presiono la tecla "q" mediante la función "cheackStop" y en caso de ser así, se finaliza la misma.

Se declara la variable cuentas, a la cual se le asigna el valor entregado por la función "setVelocidadInicial" o el valor seteado por default en caso de error.

Se declara la variable "velocidadMs", a la cual se le asigna la conversión de las cuentas del ADC a un valor porcentual de la velocidad.

Finalmente se convierte el porcentaje de velocidad calculado a un valor entero.

Si el porcentaje calculado no es válido (por ejemplo, 0), se utiliza un valor inicial definido previamente.

```
void *pageVelocidadInicial(void *arg)
{
    hilo_arg_t *args = (hilo_arg_t *)arg;
    while (1)
    {
        if (checkStop(&mut, &stop))// Verifica si se ha pulsado la tecla 'q'
            break;

        // obtener cuentas del ADC
        int cuentas = setVelocidadInicial() != -1 ? setVelocidadInicial() : 100;

        int velocidadMs = velMs(cuentas);// Calcula la velocidad en milisegundos
        cls(args->fd);// Limpia la pantalla
        dprintf(args->fd, "...Se esta configurando la velocidad inicial ...\n\n");
        // imprime el nombre de la secuencia
        dprintf(args->fd, "\t Velocidad Inicial: ---------->%.2f\n\n", velocidadMs);
        dprintf(args->fd, "Presione q para volver al menu\n\n");

        // Actualiza la velocidad
        pthread_mutex_lock(&mut);
        args->vel = velocidadMs ? velocidadMs : VEL_INICIAL;
        pthread_mutex_unlock(&mut);
    }
    stop = 0;// Actualiza la bandera para detener la secuencia
    return NULL;
}
```

6.10. Manejo de UART

En este caso se opto por un sistema en el cual una computadora se comunique con la raspberry por medio del puerto serie.

Se tendrán dos programas, el de la Raspberry Pi, el cual se encarga del procesamiento de datos, ejecución de secuencias, manejo de los hilos, etc. Mientras que el otro simplemente abre el puerto serie.

La Raspberry se encargará de configurar ambas terminales en todo momento, para la impresión en pantalla y adquisición de datos.

6.10.1. initLocal

```
int initLocal(struct termios *tty)
{
    tcgetattr(FD_STDIN, &tty[O_L_TTY]);
    tty[LOC_TTY] = tty[O_L_TTY];
    return FD_STDIN;
}
```

Inicia la terminal local y guarda su estado original en el arreglo tty, devolviendo el descriptor de archivo correspondiente.

.

6.10.2. initRemote

```
int initRemote(struct termios *tty)
{
    int fd;                                          /* Descriptor de
   archivo del puerto serie*/
    fd = open("/dev/ttyAMA0", O_RDWR | O_NOCTTY); /* Apertura del
   puerto serie */
    if (fd == -1)
    {
        printf("ERROR : no se pudo abrir el dispositivo.\n");
        return -1;
    }
    tcgetattr(fd, &tty[O_R_TTY]);
    tty[REM_TTY] = tty[O_R_TTY];
    tty[REM_TTY].c_cflag = BAUDRATE | CREAD;
    tty[REM_TTY].c_cflag &= ~PARENB;
    tty[REM_TTY].c_cflag &= ~CSTOPB;
    tty[REM_TTY].c_cflag &= ~CSIZE;
    tty[REM_TTY].c_cflag |= CS8;
    tty[REM_TTY].c_lflag |= ICANON | ECHO | ECHOE;
    tty[REM_TTY].c_cc[VTIME] = 0;          /* temporizador entre
   caracter */
    tty[REM_TTY].c_cc[VMIN] = 1;           /* bloquea lectura hasta
   llegada de un caracter */
    tcflush(fd, TCIFLUSH);                 /* Descarta datos recibidos
   en el buffer del puerto serie pero no leido aun*/
    tcsetattr(fd, TCSANOW, &tty[REM_TTY]); /* Seteo de los parametros
   de configuracion nuevos*/
    return fd;
}
```

Inicia la terminal remota y guarda su estado original en el arreglo tty, devolviendo el descriptor de archivo correspondiente. También la configura para la transmisión de datos, con un formato 8N1 sin paridad, en modo canónico.

6.10.3. setMode

```
void setMode(int *fd, struct termios *tty, int lfd, int rfd)
{
    resetCanon(tty, *fd);
    int opt_n;
    char opt;

    read(*fd, &opt, 1);
    opt_n = (int)(opt - '0');
    if (opt_n == 1)
    {
        *fd = lfd;
        dprintf(rfd, "\033[2J\033[H");
        dprintf(rfd, "MODO LOCAL\n");
        dprintf(lfd, "\033[2J\033[H");
        tty[REM_TTY].c_lflag &= ~(ECHO | ECHOE);
        tcsetattr(rfd, TCSANOW, &tty[REM_TTY]);
    }
    else if (opt_n == 2)
    {
        *fd = rfd;
        dprintf(lfd, "\033[2J\033[H");
        dprintf(lfd, "MODO REMOTO\n");
        dprintf(rfd, "\033[2J\033[H");
        tty[LOC_TTY].c_lflag &= ~(ECHO | ECHOE);
        tcsetattr(lfd, TCSANOW, &tty[LOC_TTY]);
    }
    else
    {
        *fd = -1;
    }
}
```

Toma como argumento un puntero al descriptor de archivo actual, el puntero al primer elemento del arreglo de estructuras termios, y los valores de los descriptores de archivo local y remoto respectivamente. Primero setea la terminal correspondiente a fd en modo canónico y luego espera un carácter de selección, en función de este valor se le asignará un valor a `fd` y se configuraran las terminales de manera que solo se pueden ingresar datos en la terminal seleccionada mientras que en la otra se imprime un mensaje indicando el modo seleccionado y bloqueando el ingreso de caracteres.

6.10.4. closeFD

```
void closeFD(int lfd, int rfd, struct termios *tty)
{
    resetCanon(tty, lfd);
    resetCanon(tty, rfd);
    tcflush(lfd, TCIOFLUSH);                /*Vacia buffers de entrada
   y salida*/
```

```
6     tcflush(rfd, TCIOFLUSH);
7     tcsetattr(lfd, TCSANOW, &tty[O_L_TTY]); /* Configura el puerto
      serie con los parametros originales*/
8     tcsetattr(rfd, TCSANOW, &tty[O_R_TTY]);
9     close(lfd);                             /*Cierre del puerto serie*/
10    close(rfd);
11 }
```

Toma como argumento los valores de los descriptores de archivos local y remoto, y el puntero al primer elemento del arreglo de estructuras termios. Vuelve ambas terminales a modo canónico, vacia los buffers de entrada y salida de ambas terminales, las regresa a su configuración inicial y por ultimo cierra ambos archivos.

6.11. Makefile

Con el fin de automatizar el proceso de compilación y construcción del proyecto, se realiza un archivo "Makefile". Este utilizara varias banderas y comandos que permitirá armar el proyecto de manera mas fácil. Estas banderas y comandos son:

- gcc: Compilador que se usará para el proyecto.
- -march=armv8-a: Establece arquitectura del procesador, permitiendo la ejecución de instrucciones como "MUL" y "UDIV".
- -Wall: Activa advertencias comunes para ayudarte a identificar errores.
- -Wextra: Activa advertencias adicionales, incluyendo posibles errores lógicos o prácticas de codificación cuestionables.
- -I./inc: Indica al compilador que busque archivos de cabecera adicionales en el directorio inc/.
- -c: Compila a nivel de archivo objeto, sin enlazar.
- -pthread: Habilita soporte para hilos POSIX.
- -l wiringPi: Enlaza la biblioteca WiringPi para manejar GPIO en Raspberry Pi.

```
ASFLAGS = -g -march=armv8-a
CFLAGS = -Wall -Wextra -I./inc
LDFLAGS = -lwiringPi -pthread

SRC_DIR = src
INC_DIR = inc
OBJ_DIR = obj
BIN_DIR = bin

EXEC = a

C_SRCS = $(wildcard $(SRC_DIR)/*.c)
AS_SRCS = $(wildcard $(SRC_DIR)/*.s)
OBJS = $(patsubst $(SRC_DIR)/%.c,$(OBJ_DIR)/%.o,$(C_SRCS)) $(patsubst $
    (SRC_DIR)/%.s,$(OBJ_DIR)/%.o,$(AS_SRCS))

TARGET = $(BIN_DIR)/$(EXEC)

all: $(TARGET)

$(TARGET): $(OBJS) | $(BIN_DIR)
  $(CC) $(OBJS) -o $@ $(LDFLAGS)

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c | $(OBJ_DIR)
  $(CC) $(CFLAGS) -c $< -o $@

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.s | $(OBJ_DIR)
  $(AS) $(ASFLAGS) $< -o $@

$(BIN_DIR):
  mkdir -p $(BIN_DIR)

```

```
32
33 $(OBJ_DIR):
34   mkdir -p $(OBJ_DIR)
35
36 clean:
37   rm -rf $(OBJ_DIR) $(BIN_DIR)
38
39 .PHONY: all clean
```

Como se puede ver el "Makefile" esta construido para que trabaje sobre todos los archivos que están en carpetas. De esta manera se se organiza mejor el proyecto y hace mas fácil la expansión en caso de ser necesario.

7. Imágenes de consola

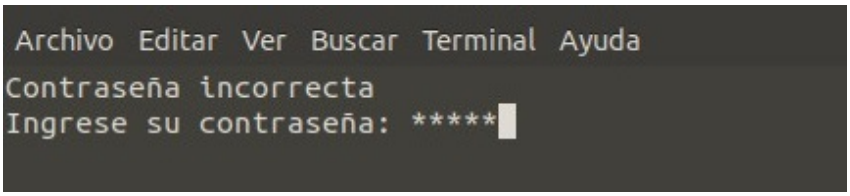


Figura 4: Consola en apartado de autenticación

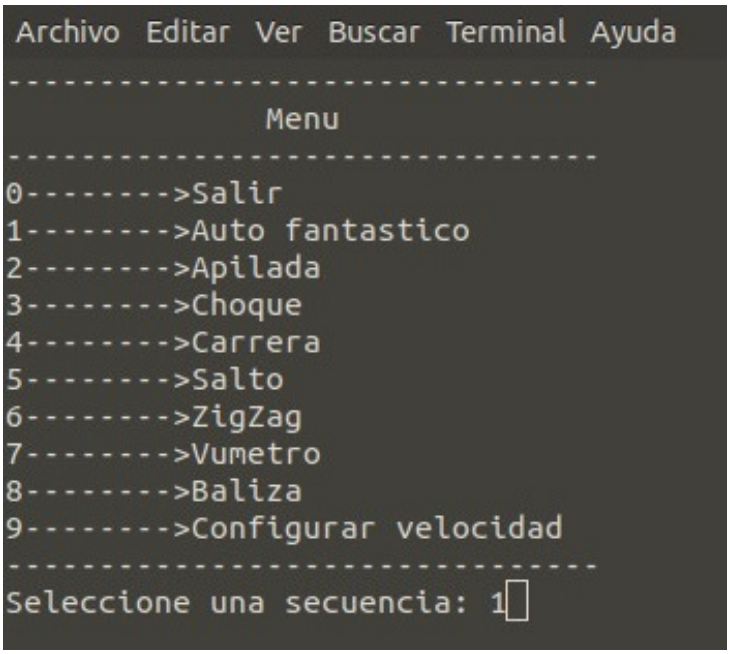


Figura 5: Consola en menú principal

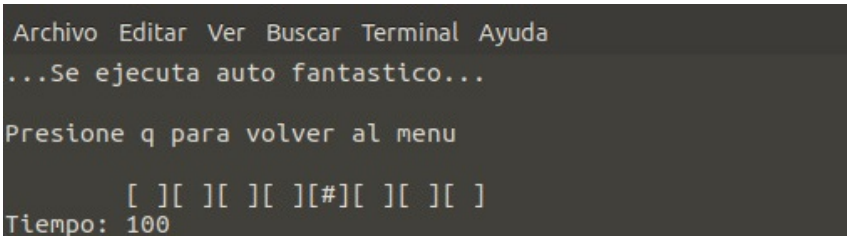


Figura 6: Consola en secuencia de auto fantástico

8. Conclusión

El desarrollo de este proyecto permitió implementar un sistema interactivo y multifuncional basado en una Raspberry Pi, cumpliendo con los requisitos planteados inicialmente. Mediante el uso de WiringPi y pthreads, se lograron diseñar e implementar ocho secuencias de luces LED con opciones de configuración avanzadas y una interfaz de usuario eficiente y amigable.

El sistema demostró ser robusto al incorporar un mecanismo de control de acceso mediante contraseña, garantizando seguridad en el uso del menú de opciones. Además, se implementaron técnicas que permitieron una interacción dinámica, como el ajuste inmediato de la velocidad de las secuencias, la visualización en tiempo real de la secuencia activa y la capacidad de interrumpir la ejecución para regresar al menú principal.

El uso del conversor A/D para inicializar las velocidades de las secuencias a partir de potenciómetros conectados al hardware no solo fortaleció la integración hardware-software, sino que también potenció la personalización del sistema desde el inicio de su ejecución. Asimismo, la implementación de modos de operación local y remoto mediante comunicación UART amplió las capacidades del programa, adaptándolo a diferentes escenarios de hardware y mostrando la versatilidad del diseño.

La inclusión de una función escrita en ensamblador ARM evidenció el dominio de técnicas de programación de bajo nivel y su integración con herramientas de alto nivel, lo que permitió optimizar y diversificar las capacidades del sistema. Por otra parte, la combinación de algoritmos personalizados y tablas de datos para las secuencias personalizadas demostró la eficacia de diferentes enfoques de programación en contextos específicos.

En conclusión, el proyecto alcanzó sus objetivos planteados al inicio, ofreciendo una solución completa, funcional y adaptable. Este trabajo no solo permitió afianzar conocimientos teóricos y prácticos sobre programación concurrente, control de hardware y comunicación serial, sino que también resaltó la importancia de la planificación y el diseño modular en el desarrollo de sistemas complejos.

Referencias

[1] Hoja de datos ADC 1115: ttps://www.ti.com/lit/ds/symlink/ads1115.pdf?ts=1733784922815&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252Fes-mx%252FADS1115

[2] Archivos fuente del proyecto: https://github.com/Gomez-Enzo/Proyecto_Final_Tecnicas_2