Microcontroladores: Laboratorio 2

1st Hector Pereira
Ingeniería en Mecatrónica
Universidad Tecnológica (UTEC)
Fray Bentos, Uruguay
hector.pereira@estudiantes.utec.edu.uy

2nd Isaac Martirena
Ingeniería en Mecatrónica
Universidad Tecnológica (UTEC)
Fray Bentos, Uruguay
isaac.martirena@estudiantes.utec.edu.uy

***Resumen*—**
Keywords:

IV-A2. Colores: El proceso de reconocimiento de colores se fundamenta en la descomposición de cada color en sus componentes primarias dentro del modelo RGB (*Red*, *Green*, *Blue*). Un color puede representarse como la combinación ponderada de estas tres intensidades, lo que permite su descripción en un espacio tridimensional.

Sin embargo, el sensor fotoresistivo (LDR, *Light Dependent Resistor*) utilizado en el sistema no distingue de forma individual las componentes cromáticas del espectro visible; únicamente mide la intensidad total de luz reflejada sobre su superficie. Si la iluminación se realiza con luz blanca, el sensor solo entrega una lectura proporcional a la cantidad total de luz reflejada, sin discriminar su composición espectral. Este fenómeno equivale a una percepción en escala de grises, dificultando la identificación precisa de colores.

Para resolver esta limitación, se empleó un diodo emisor de luz RGB como fuente de iluminación controlada. Al iluminar secuencialmente la superficie con luz roja, verde y azul, el sistema obtiene tres mediciones independientes mediante el LDR. Dichos valores, convertidos por el módulo ADC (*Analog-to-Digital Converter*) del microcontrolador ATmega328P, representan las coordenadas (R, G, B) de un punto dentro de un espacio de color tridimensional.

Dado que tanto la respuesta espectral del LDR como la emisión de los LED presentan variaciones no lineales, los valores obtenidos no son directamente proporcionales a las componentes RGB teóricas. No obstante, el sistema logra resultados estables mediante un proceso de calibración inicial, donde se iluminan sucesivamente los colores de referencia (rojo, verde, azul claro, violeta, morado, amarillo y blanco) y se almacenan sus valores de conversión analógica-digital.

Cada color de referencia calibrado se modela como un vector fijo en dicho espacio. Para identificar un color, se calcula la distancia euclidiana entre el vector de medición y cada uno de los vectores de referencia almacenados:

$$D = \sqrt{(R_m - R_i)^2 + (G_m - G_i)^2 + (B_m - B_i)^2} \quad (1)$$

donde (R_m, G_m, B_m) representan los valores medidos

por el sensor y (R_i, G_i, B_i) corresponden a los valores calibrados de cada color de la hoja de referencia. El color identificado será aquel cuya distancia D sea mínima.

El circuito se implementó mediante un divisor resistivo conectado a una de las entradas analógicas del microcontrolador. El color identificado se replica visualmente en la tira de LED WS2812, y el servomotor se posiciona en el ángulo correspondiente al color detectado dentro de la hoja de referencia, integrando así un sistema de selección e identificación de color completamente automatizado.

IV-A3. Piano: El sonido es una onda mecánica que se propaga a través de un medio elástico, producto de variaciones periódicas de presión. Estas ondas pueden clasificarse según su forma en senoidales, cuadradas, triangulares o diente de sierra, dependiendo del patrón temporal de oscilación. En los sistemas digitales, las señales más sencillas de generar son las ondas cuadradas, donde el voltaje alterna entre dos niveles definidos, representando los estados lógicos alto y bajo del microcontrolador.

Para producir sonido de manera electrónica, se emplean dispositivos piezoeléctricos denominados *buzzers*. Estos transductores convierten la energía eléctrica en vibraciones mecánicas audibles. Cuando el microcontrolador aplica una señal cuadrada a la entrada del buzzer, el material piezoeléctrico se deforma y contrae periódicamente, generando un sonido cuya frecuencia está directamente relacionada con la frecuencia de la señal de entrada.

En el microcontrolador ATmega328P, esta señal se genera mediante el uso de los temporizadores internos (*timers*), configurados para producir una onda cuadrada en un pin de salida. Al modificar la frecuencia de conmutación, es posible variar el tono percibido, ya que la frecuencia del sonido determina su altura musical. De esta manera, cada nota puede asociarse a una frecuencia específica según la escala temperada. Por ejemplo, la nota *La4* corresponde a una frecuencia de 440 Hz, mientras que *Do4* equivale aproximadamente a 262 Hz.

Una nota musical puede representarse computacionalmente mediante dos parámetros fundamentales: la frecuencia de oscilación, la duración temporal durante la cual se mantiene activa y la duración inactiva o en silencio. Cuando el buzzer emite una secuencia ordenada de notas, cada una con su tiempo de activación y silencio, se obtiene una melodía. En términos programáticos, una melodía se define como un conjunto de estructuras de datos que contienen pares `(frecuencia, tiempo_on, tiempo_off)`, los cuales son procesados secuencialmente para generar la música deseada.

La reproducción simultánea de varios instrumentos en un entorno digital requiere el manejo de múltiples secuencias de notas en paralelo. Para lograrlo sin recurrir al uso de ciclos de espera activos (*polling*), se utilizan interrupciones asociadas a los temporizadores. De este modo, el microcontrolador puede controlar la duración y el inicio de cada nota de forma independiente y precisa, optimizando el uso del procesador y permitiendo la ejecución de tareas concurrentes, como la lectura de entradas o la comunicación serial, mientras la música continúa sonando de manera autónoma.

En síntesis, el sistema desarrollado utiliza un buzzer piezoeléctrico controlado por los temporizadores del ATmega328P para reproducir notas musicales definidas por su frecuencia y duración. A través de la programación de secuencias almacenadas en memoria, es posible interpretar melodías completas y gestionar la reproducción de distintas canciones sin intervención del usuario durante la ejecución.

IV-A4. Cerradura: El sistema de cerradura electrónica desarrollado tiene como objetivo controlar el acceso mediante una contraseña numérica almacenada en memoria no volátil. Inicialmente, el dispositivo se encuentra en estado de *candado cerrado*. Cuando el usuario ingresa la contraseña correcta, el sistema cambia al estado de *candado abierto*, permitiendo el acceso. En caso de introducir una contraseña incorrecta tres veces consecutivas, se activa una alarma acústica a través de un buzzer, indicando un intento de acceso no autorizado.

El sistema también permite modificar la contraseña almacenada. Para ello, el usuario debe ingresar primero la contraseña actual y, tras su validación, definir una nueva contraseña de entre cuatro y seis dígitos. Esta nueva clave se almacena permanentemente en la memoria EEPROM del microcontrolador, garantizando su persistencia incluso después de un apagado o reinicio del sistema.

Interfaz de usuario e interacción: La cerradura dispone de una interfaz de usuario compuesta por una pantalla LCD de 16x2, un teclado matricial 4x4 y tres indicadores luminosos (LED verde, LED rojo y alarma sonora). En este contexto, la interfaz constituye el medio de comunicación entre el usuario y el sistema, mostrando mensajes informativos y recibiendo acciones por medio del teclado. Cada interacción del usuario genera una respuesta visual o acústica distinta, representando así un flujo de diálogo entre ambos.

El sistema se diseñó siguiendo una lógica de *máquina de estados finitos*, en la que cada modo de operación (menú principal, ingreso de contraseña, cambio de clave, acceso autorizado, alarma, etc.) representa un estado. Las transiciones entre estados se producen en función de las acciones del usuario y las condiciones del sistema. Este enfoque facilita la gestión de comportamientos complejos, simplifica el control del flujo de ejecución y mejora la legibilidad del código.

Teclado matricial y detección de teclas: El ingreso de datos se realiza mediante un teclado matricial 4x4 que combina filas y columnas, optimizando el uso de pines del microcontrolador. Las filas se configuran como salidas y las columnas como entradas con resistencias de *pull-up* activadas. El proceso de lectura consiste en activar una fila a nivel bajo (0) mientras las demás permanecen en alto (1); si

alguna tecla de esa fila se encuentra presionada, la columna correspondiente cambia a nivel bajo, permitiendo identificar la intersección entre fila y columna.

Cada tecla se asocia a un carácter según su posición dentro de la matriz, lo que permite retornar el valor numérico o simbólico correspondiente. Para evitar falsas detecciones debido al rebote mecánico de los contactos, se aplica una rutina de *debouncing* por software basada en pequeños retardos temporizados.

Gestión temporal y concurrencia de tareas: Durante el funcionamiento, el sistema debe ejecutar múltiples tareas de forma paralela: reproducción de sonidos, manejo de alarmas, parpadeo de LEDs, lectura del teclado y actualización de la pantalla LCD. Sin embargo, el ATmega328P dispone de un número limitado de temporizadores, por lo que no es posible asignar un temporizador independiente a cada tarea.

Para resolver esta limitación, se implementó un esquema de ejecución basado en un *planificador de tareas* o *task scheduler* de propósito general. Se utiliza un solo temporizador configurado para generar interrupciones periódicas, incrementando un contador global de 32 bits que actúa como referencia temporal (en milisegundos). Cada tarea compara el tiempo actual con el instante programado de su próxima ejecución, y si se cumple el intervalo, se ejecuta la acción correspondiente. De este modo, múltiples eventos temporizados pueden coexistir sin emplear retardos bloqueantes ni técnicas de *polling*.

Almacenamiento persistente de la contraseña: Para asegurar que la contraseña permanezca almacenada tras un apagado, se utiliza la memoria EEPROM interna del ATmega328P. Esta memoria no volátil permite conservar los datos durante décadas sin alimentación eléctrica, aunque posee un número limitado de ciclos de escritura (aproximadamente 10^5 operaciones por celda). Por ello, el sistema únicamente escribe en EEPROM cuando el usuario confirma un cambio de contraseña, minimizando el desgaste de la memoria.

La lectura y escritura se realizan mediante las funciones de la biblioteca estándar `avr/eeprom.h`, que permiten transferir cadenas de caracteres directamente desde y hacia direcciones específicas de la EEPROM. Así, el sistema recupera la contraseña almacenada al inicio del programa y la compara con la ingresada por el usuario durante la operación normal.

Resumen de funcionamiento: En conjunto, la cerradura electrónica combina la interacción mediante teclado y pantalla LCD, la gestión de tareas en tiempo real y el almacenamiento persistente de datos, logrando un sistema confiable y autónomo. El uso de una máquina de estados y un planificador temporal basado en un único temporizador permite controlar de manera eficiente múltiples procesos simultáneos sin bloquear la ejecución principal del programa.

IV-B. Evidencias gráficas y mediciones

IV-C. Comentario general

V. Conlusiones

V-A. Plotter

V-B. Colores

- La comunicación por USART fue una limitación para la velocidad del sistema. Podría mejorarse reduciendo la cantidad de datos enviados o usando interrupciones para hacerlo más fluido.
- Sería útil agregar un modo de calibración donde el usuario pueda registrar los valores de referencia de cada color, adaptando el sistema a distintas condiciones de luz.
- Se podría usar un servomotor de 360° para aprovechar todos los colores disponibles en la rueda.

V-C. Piano

- Sería interesante agregar más canciones al repertorio.
- Se podría mejorar la coordinación entre las pistas, ya que con el tiempo tienden a desfasarse un poco.
- También sería bueno incluir más notas en el teclado para ampliar el rango musical.

V-D. Cerradura

- Agregar un botón para ver la contraseña escrita antes de confirmar ayudaría al usuario sin afectar la seguridad.
- Implementar un solenoide real permitiría que el sistema abra y cierre físicamente el cerrojo.
- En la simulación de PicSimLab el reinicio no siempre funciona correctamente. A veces es necesario usar el botón “DEBUG” o reiniciar el programa. Sin embargo, en el dispositivo real el sistema funciona correctamente.

Referencias

[1] Microchip Technology Inc. Atmega328p datasheet. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[2] circuito.io. (2018) Arduino uno pinout diagram. [Online]. Available: https://www.circuito.io/blog/arduino-uno-pinout/

[3] ARXterra. Addressing modes — 8-bit avr instruction set. [Online]. Available: https://www.arxterra.com/3-addressing-modes/

[4] Software Particles. (2024, Apr.) Learn how an 8×8 led display works and how to control it using an arduino. Figura: 8×8 LED matrix pin mapping (imagen del artículo). [Online]. Available: https://softwareparticles.com/learn-how-an-8x8-led-display-works-and-how-to-control-it-using-an-arduino/

[5] Carpeta del laboratorio (google drive). Carpeta compartida del laboratorio. [Online]. Available: https://drive.google.com/drive/u/0/folders/1fP0aILozXeapRgDPDNWT1TRhAYr1PPPT

[6] Microchip Community (AVR Freaks). Avr freaks — comunidad de desarrolladores avr. Foros técnicos y soluciones prácticas sobre AVR. [Online]. Available: https://www.avrfreaks.net/

[7] J. Ganssle. A guide to debouncing. Referencia clásica para anti-rebote en pulsadores/encoders. [Online]. Available: https://www.ganssle.com/debouncing.htm

[8] G. Schmidt. (2021, Sep.) Beginners introduction to the assembly language of atmel-avr-microprocessors. Tutorial de avr-asm-tutorial.net (revisión septiembre 2021). [Online]. Available: https://kitsandparts.com/tutorials/assemblers/BeginnersAVRasm.pdf

[9] T. Redelberger. (2019, Apr.) Avr assembler for complex projects. Versión 0.4 (2019-04-06). [Online]. Available: https://web222.webclient5.de/doc/swdev/avrasm/advavrasm2/AdvancedUseAVRASM2_en_20190406.pdf
[10] Laboratorio de Microcontroladores, UTEC, "Repositorio de laboratorio de microcontroladores (tec.micro)," https://github.com/MateoLecuna/Tec.Micro, 2025, accedido el 26 de septiembre de 2025.

VI. Anexos

VI-A. Fragmentos de codigo

VI-A1. Plotter: —

VI-A2. Colores: —

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
```

Listing 1. Librerias utilizadas

```
typedef struct {
    const char *name;
    uint16_t r, g, b;
} ColorRef;

const ColorRef color_refs[] = {
    {"MORADO",    216, 157, 274},
    {"ROJO",       206, 149, 262},
    {"AMARILLO",    196, 99, 105},
    {"VERDE",     272, 192, 152},
    {"AZUL CLARO", 151, 153, 122},
    {"VIOLETA",    253, 275, 338},
    {"BLANCO",    110, 93, 91},
};
```

Listing 2. Mapeado de colores

```
// Convierte un valor entero sin signo en un
    string
void UTOA(uint16_t value, char *buffer) {
    char temp[6];
    int i = 0, j = 0;

    if (value == 0) {
        buffer[0] = '0';
        buffer[1] = '\0';
        return;
    }

    // Convert digits to temp buffer (reversed)
    while (value > 0 && i < sizeof(temp) - 1) {
        temp[i++] = (value % 10) + '0';
        value /= 10;
    }

    // Reverse digits into final buffer
    while (i > 0) buffer[j++] = temp[--i];
    buffer[j] = '\0';
}
```

Listing 3. Convertidor de unsigned integer a string

```
void send_bit(uint8_t bitVal){
    if(bitVal){
        PORTD |=  (1<<LED_PIN);
        asm volatile (
        "nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n
    \t"
        "nop\n\t""nop\n\t""nop\n\t""nop\n\t");

        PORTD &= ~(1<<LED_PIN);

        asm volatile (
        "nop\n\t""nop\n\t""nop\n\t""nop\n\t");

        } else {
        PORTD |=  (1<<LED_PIN);
        asm volatile (
        "nop\n\t""nop\n\t""nop\n\t");

        PORTD &= ~(1<<LED_PIN);
        asm volatile (
        "nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n
    \t"
        "nop\n\t""nop\n\t""nop\n\t""nop\n\t""nop\n
    \t");
    }
}

// Envia un Byte a la tira de leds
// Utiliza cli y sei para un timing preciso
void send_byte(uint8_t byte) {
    cli();
    for (uint8_t i = 0; i < 8; i++) {
        send_bit(byte & 0x80);  // send most
    significant bit first
        byte <<= 1;             // shift next bit
    into MSB position
    }
    sei();  // re-enable interrupts
}
```

Listing 4. Enviado de datos a tira LED

```
void usart_init(void) {
    const uint16_t ubrr = (16000000UL / (16UL *
    BAUD_RATE)) - 1;
    UBRR0H = ubrr >> 8;
    UBRR0L = ubrr;
    UCSR0A = 0;
    UCSR0B = (1 << TXEN0) | (1 << RXEN0) | (1 <<
    RXCIE0);   // RX interrupt
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
            // 8N1
}

void adc_init(void) {
    ADMUX  = (1 << REFS0);
    ADCSRA = (1 << ADEN)
    | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
    // Prescaler 128
}

void rgb_init(void){
    DDRB |= (1<<RED) | (1<<GREEN) | (1<<BLUE);
}


void servo_init(void) {
    SERVO_DDR |= (1 << SERVO_PORT);

    TCCR1A = (1 << COM1A1) | (1 << WGM11);
    TCCR1B = (1 << WGM13) | (1 << WGM12) | (1 <<
    CS11); // 8

    ICR1 = 39999;
}

void ws2812_init(void) { // tira de leds
    LED_DDR |= (1 << LED_PIN);
}
```

Listing 5. Inicializadores

```
void ws2812_send_pixel(uint8_t r, uint8_t g,
    uint8_t b) {
    // Cambiar orden dependiendo de formato de
    tira led o matriz
    send_byte(g);
    send_byte(b);
    send_byte(r);
}

// Encender n leds del mismo color
void ws2812_fill(uint8_t r, uint8_t g, uint8_t b,
    uint16_t n) {
    cli();
    for (uint16_t i = 0; i < n; i++) {
        ws2812_send_pixel(r, g, b);
    }
    sei();
    ws2812_show();
}

// Establecer colores predeterminados
void led_strip_set_color(uint8_t color_id) {
    uint8_t r = 0, g = 0, b = 0;

    switch (color_id) {
        case 1: // Rojo
        r = 255; g = 0; b = 0;
        break;

        case 2: // Amarillo
        r = 255; g = 100; b = 0;
        break;

        case 3: // Verde
        r = 0; g = 255; b = 0;
        break;

        case 4: // Azul claro
        r = 0; g = 255; b = 255;
        break;

        case 5: // Violeta
        r = 100; g = 0; b = 100;

        break;

        case 6: // Morado
        r = 200; g = 0; b = 75;

        break;

        case 7: // Blanco
        r = 255; g = 255; b = 255;
        break;

        default: // Apagar LED
        r = g = b = 0;
        break;
    }

    ws2812_fill(r, g, b, 50);
}

// Establecer angulo en el servomotor
void servo_set_angle(uint8_t angle) {
    uint16_t pulse = 1000 + ((uint32_t)angle *
    4000) / 180;
    OCR1A = pulse;
}
```

```
// Leer adc
uint16_t adc_read(uint8_t channel) {
    ADMUX = (ADMUX & 0xF0) | (channel & 0x0F);
    ADCSRA |= (1 << ADSC);
    while (ADCSRA & (1 << ADSC));               //
     Wait for conversion to finish
    return ADC;
}

// Establecer color del led rgb (iluminador)
void rgb_set(uint8_t r, uint8_t g, uint8_t b) {
    PORTB = (PORTB & ~((1 << RED)|(1 << GREEN)|(1
    << BLUE))) |
    ((r<<RED) | (g<<GREEN) | (b<<BLUE));
}

// Identificar color a partir de valores rgb.
// Tomando cada valor calibrado como un vector, se
     determina la distancia
// cartesiana del vector leido por el sensor.
const char* identify_color(uint16_t r, uint16_t g,
     uint16_t b) {
    uint32_t best_dist = 0xFFFFFFFF;
    const char *best_name = "UNKNOWN";

    for (uint8_t i = 0; i < NUM_COLORS; i++) {
        int32_t dr = (int32_t)r - color_refs[i].r;
        int32_t dg = (int32_t)g - color_refs[i].g;
        int32_t db = (int32_t)b - color_refs[i].b;
        uint32_t dist = dr*dr + dg*dg + db*db;

        if (dist < best_dist) {
            best_dist = dist;
            best_name = color_refs[i].name;
        }
    }
    return best_name;
}
```

Listing 6. Funciones de utilidad

```
// programa principal
int main(void) {
    ws2812_init();
    usart_init();
    adc_init();
    rgb_init();
    servo_init();
    sei();

    while (1) {
        rgb_read();
        _delay_ms(50);


    }
}
```

Listing 7. Bucle principal

VI-A3. Piano: —
USART con buffers

```
#define TX_BUF_SZ 128
#define TX_MASK   (TX_BUF_SZ - 1)

#define RX_BUF_SZ 128
#define RX_MASK   (RX_BUF_SZ - 1)

uint8_t tx_buf[TX_BUF_SZ];
uint8_t tx_head = 0, tx_tail = 0;
```

```
uint8_t rx_buf[RX_BUF_SZ];
uint8_t rx_head = 0, rx_tail = 0;

uint8_t usart_rx_available(void) {
    return (uint8_t)((rx_head - rx_tail) & RX_MASK
    );
}

void usart_init_9600(void) {
    const uint16_t ubrr = (16000000UL / (16UL *
    9600)) - 1;
    UBRR0H = ubrr >> 8;
    UBRR0L = ubrr;
    UCSR0A = 0;
    UCSR0B = _BV(TXEN0) | _BV(RXEN0) | _BV(RXCIE0)
    ;   // <- RX interrupt
    UCSR0C = _BV(UCSZ01) | _BV(UCSZ00);
        // 8N1
}

// USART
void usart_init_9600(void) {
    const uint16_t ubrr = (16000000UL / (16UL *
    9600)) - 1;
    UBRR0H = ubrr >> 8;
    UBRR0L = ubrr;
    UCSR0A = 0;
    UCSR0B = _BV(TXEN0) | _BV(RXEN0) | _BV(RXCIE0)
    ;   // <- RX interrupt
    UCSR0C = _BV(UCSZ01) | _BV(UCSZ00);
        // 8N1
}

// Escribir byte al buffer
uint8_t usart_write_try(uint8_t b) {
    uint8_t next = (uint8_t)((tx_head + 1) &
    TX_MASK);
    if (next == tx_tail) return 0;
    // buffer lleno
    tx_buf[tx_head] = b;
    tx_head = next;
    UCSR0B |= _BV(UDRIE0);
    // kick the ISR
    return 1;
}

// Escribir string al buffer
uint16_t usart_write_str(const char *s) {
    uint16_t n = 0;
    while (*s && usart_write_try((uint8_t)*s++)) n
    ++;
    return n;
}

// Leer byte del buffer de recepcion
uint8_t usart_read_try(uint8_t *b) {
    if (rx_head == rx_tail) return 0;
    *b = rx_buf[rx_tail];
    rx_tail = (uint8_t)((rx_tail + 1) & RX_MASK);
    return 1;
}

// Interrupcion de registro de TX libre
ISR(USART_UDRE_vect) {
    if (tx_head == tx_tail) {
        UCSR0B &= (uint8_t)~_BV(UDRIE0);
        return;
    }
    UDR0 = tx_buf[tx_tail];
    tx_tail = (uint8_t)((tx_tail + 1) & TX_MASK);
}

// Interrupcion de dato recibido
ISR(USART_RX_vect) {
    uint8_t d = UDR0;
    uint8_t next = (uint8_t)((rx_head + 1) &
    RX_MASK);
    if (next != rx_tail) {
        rx_buf[rx_head] = d;
        rx_head = next;
    }
}
```

Listing 8. Funciones USART con buffering

Formato de guardado de canciones

```
// Midi tracks
// Generated using https://github.com/ShivamJoker/
    MIDI-to-Arduino
const int midiC[827][3] PROGMEM = {
    {E5, 94, 0},
    {B4, 94, 0},
    {A4, 94, 0},
    {E4, 94, 0},
    {A4, 94, 0},
    {B4, 94, 0}
    // ...
}
```

Listing 9. Guardado de canciones

```
void handleUSART(uint8_t character){
    if (character == '1'){
        mode = 1;
        eventAoff = 1;
        eventBoff = 0;

        song = 0;

        eventAon = 0; // Encender track A
        indexA = 0; // Posicion track A
        countA = 0; // Conteo de overflow de notas
     de track A
        maxCountAon = 0; // Maximo conteo de
    overflow encendido en A
        maxCountAoff = 0; // Maximo conteo de
    overflow apagado en A
        enableCountAon = 0; // Habilitar conteo de
     encendido en A
        enableCountAoff = 0; // Habilidad conteo
    de apagado en A

        eventBon = 0; // Encender track B
        indexB = 0; // Posicion track B
        countB = 0; // Conteo de overflow de notas
     de track B
        maxCountBon = 0; // Maximo conteo de
    overflow encendido en B
        maxCountBoff = 0; // Maximo conteo de
    overflow apagado en B
        enableCountBon = 0; // Habilitar conteo de
     encendido en B
        enableCountBoff = 0; // Habilidad conteo
    de apagado en B

        PCICR &= ~((1 << PCIE1) | (1 << PCIE2));
        stopFrequencyB();

    } else if (character == '2'){
        mode = 1;
        eventAoff = 1;
        eventBoff = 1;

        song = 1;

        eventAon = 0; // Encender track A
        indexA = 0; // Posicion track A
        countA = 0; // Conteo de overflow de notas
```

```
     de track A
        maxCountAon = 0; // Maximo conteo de
    overflow encendido en A
        maxCountAoff = 0; // Maximo conteo de
    overflow apagado en A
        enableCountAon = 0; // Habilitar conteo de
     encendido en A
        enableCountAoff = 0; // Habilidad conteo
    de apagado en A

        eventBon = 0; // Encender track B
        indexB = 0; // Posicion track B
        countB = 0; // Conteo de overflow de notas
     de track B
        maxCountBon = 0; // Maximo conteo de
    overflow encendido en B
        maxCountBoff = 0; // Maximo conteo de
    overflow apagado en B
        enableCountBon = 0; // Habilitar conteo de
     encendido en B
        enableCountBoff = 0; // Habilidad conteo
    de apagado en B

        PCICR &= ~((1 << PCIE1) | (1 << PCIE2));
        stopFrequencyA();

    } else if (character == 'P'){
        mode = 0;
        eventAoff = 0;
        eventBoff = 0;

        eventAon = 0; // Encender track A
        indexA = 0; // Posicion track A
        countA = 0; // Conteo de overflow de notas
     de track A
        maxCountAon = 0; // Maximo conteo de
    overflow encendido en A
        maxCountAoff = 0; // Maximo conteo de
    overflow apagado en A
        enableCountAon = 0; // Habilitar conteo de
     encendido en A
        enableCountAoff = 0; // Habilidad conteo
    de apagado en A

        eventBon = 0; // Encender track B
        indexB = 0; // Posicion track B
        countB = 0; // Conteo de overflow de notas
     de track B
        maxCountBon = 0; // Maximo conteo de
    overflow encendido en B
        maxCountBoff = 0; // Maximo conteo de
    overflow apagado en B
        enableCountBon = 0; // Habilitar conteo de
     encendido en B
        enableCountBoff = 0; // Habilidad conteo
    de apagado en B

        startDebounceTimer();
        stopFrequencyA();
        stopFrequencyB();
    }
}
```

Listing 10. Manejo de eventos RX USART

```
void playFrequencyA(uint16_t freq) {
    if (!freq) return;

    DDRD |= (1 << PORTD6);

    // Elegir prescaler adecuado para esa
    frecuencia

    uint8_t presc_bits = 0;
    uint16_t ocr = 0;

    const uint16_t presc_list[] = {8, 64, 256,
    1024};
    const uint8_t  bits_list[]  = {0b010, 0b011, 0
    b100, 0b101};

    for (uint8_t i = 0; i < 4; i++) {
        ocr = (F_CPU / (2UL * presc_list[i] * freq
    )) - 1;
        if (ocr <= 255) {
            presc_bits = bits_list[i];
            break;
        }
    }

    TCCR0A = (1 << COM0A0) | (1 << WGM01);
    TCCR0B = presc_bits;
    OCR0A  = (uint8_t)ocr;
}
```

Listing 11. Reproduccion de frecuencias

```
uint8_t eventAon = 0; // Encender track A
uint8_t eventAoff = 0; // Apagar track A
uint16_t indexA = 0; // Posicion track A
uint16_t countA = 0; // Conteo de overflow de
    notas de A
uint16_t maxCountAon = 0; // Maximo conteo de
    overflow encendido en A
uint16_t maxCountAoff = 0; // Maximo conteo de
    overflow apagado en A
uint8_t enableCountAon = 0; // Habilitar conteo de
     encendido en A
uint8_t enableCountAoff = 0; // Habilidad conteo
    de apagado en A

uint8_t eventBon = 0; // Encender track B
uint8_t eventBoff = 0; // Apagar track B
uint16_t indexB = 0; // Posicion track B
uint16_t countB = 0; // Conteo de overflow de
    notas de B
uint16_t maxCountBon = 0; // Maximo conteo de
    overflow encendido en B
uint16_t maxCountBoff = 0; // Maximo conteo de
    overflow apagado en B
uint8_t enableCountBon = 0; // Habilitar conteo
    encendido en B
uint8_t enableCountBoff = 0; // Habilitar conteo
    apagado en B
```

Listing 12. Variables de control de flujo

```
int main(void) {
    timer1_init();
    usart_init_9600();
    init_piano_buttons();
    sei();

    usart_write_str("Elija una opcion:\r\n");

    usart_write_str(
        "[1] Dragon Ball - Cha-La Head-Cha-La\r\n"
        );

    usart_write_str(
        "[2] Portal - Still alive\r\n"
        );

    usart_write_str("[P] Modo piano\r\n");


    while (1){
```

```
        if (mode == 0){
            piano_mode();
        } else if (mode == 1){
            song_mode();
        }
        uint8_t c;
        if (usart_read_try(&c)) {
            handleUSART(c);
        }
    }
}
```

Listing 13. Variables de control de flujo

VI-A4. Cerradura: — Diagrama de Flujo

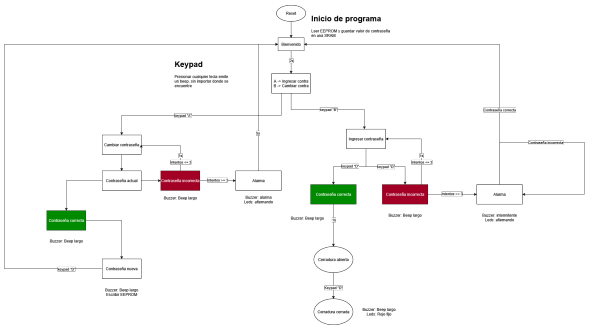

Figura 1. Diagrama de flujo. Fuente: elaboración própia

Librerias para lcd

```
#include "i2c_master.h"
#include "i2c_master.c"
#include "liquid_crystal_i2c.h"
#include "liquid_crystal_i2c.c"

int main(void){
    LiquidCrystalDevice_t device = lq_init(0x27,
    16, 2, LCD_5x8DOTS);
    lq_turnOnBacklight(&device);
    char welcomeText[] = "Bienvenido!";
    lq_print(&device, welcomeText);
    while (1);
}
```

Listing 14. Librerias utilizadas para pantalla LCD

Implementacion de tareas

```
uint32_t millis_counter = 0;

// Mapeo de keypad
const char keypad[4][4] = {
    {'1', '2', '3', 'A'},
    {'4', '5', '6', 'B'},
    {'7', '8', '9', 'C'},
    {'*', '0', '#', 'D'}
};


uint32_t keypad_on_at = 0;
uint8_t keypad_enable = 1;

ISR(TIMER0_OVF_vect){
    millis_counter++;
}

uint32_t millis_now(void) {
    uint32_t m;
    cli();      // disable interrupts
    m = millis_counter;
    sei();      // re-enable
    return m;
}

void keypad_debounce_ms(uint16_t delay_ms){
    keypad_enable = 0;
    keypad_on_at = millis_now() + delay_ms;
}

void keypad_task(void){
    if (!keypad_enable && (millis_now() >
    keypad_on_at)){
        keypad_enable = 1;
    }
}

int main(void){
    // main code ...
    while (1){
        keypad_task();
        // loop code ...
    }
}
```

Listing 15. Implementacion de tareas

Lectura multiplexada de keypad

```
char keypad_scan(void) {
    if (!keypad_enable) return 0;

    uint8_t row, col;
    uint8_t cols;
    static uint8_t prevKey; // store for later

    for (row = 0; row < 4; row++) {
        PORTD = (PORTD | 0xF0) & ~(1 << (row + 4))
    ;
        _delay_us(5);
        cols = PIND & 0x0F;

        for (col = 0; col < 4; col++) {
            if (!(cols & (1 << col)) ) {
                if ((prevKey == keypad[row][col]))
     return 0;

                keypad_debounce_ms(200);
                prevKey = keypad[row][col];
                return keypad[row][col];
            }
        }
    }
    prevKey = 0;
    return 0;
}
```

Listing 16. Funcion de lectura multiplexada de keypad

Tarea de alarma

```
 void alarm_task(void){
    if (!alarm_active) return;
    uint32_t now = millis_now();

    if (now > alarm_until){
        alarm_active = 0;
        PORTB &= ~(1<<PORTB5);
        led_red_on();
        return;
    }
```

```
    if (now > alarm_next_toggle){
        alarm_next_toggle = now + ALARM_TOGGLE_MS;
        if (alarm_phase){
            led_red_on();
        } else {
            led_green_on();
        }
        alarm_phase ^= 1;

        buzzer_beep(100);
    }
}
```

Listing 17. Tarea de alarma

Escritura y lectura de EEPROM

```
#include <avr/eeprom.h>

#define MAX_PASSWORD_LENGTH 6
#define EEPROM_MAGIC 0x42

uint8_t EEMEM ee_magic;
char EEMEM ee_password[MAX_PASSWORD_LENGTH + 1];

char storedPassword[MAX_PASSWORD_LENGTH + 1] = "
    123456";
char typedPassword[MAX_PASSWORD_LENGTH + 1];


void eeprom_load_password(void) {
    if (eeprom_read_byte(&ee_magic) !=
    EEPROM_MAGIC) {

    eeprom_update_block(
            storedPassword,
            ee_password,
            sizeof(storedPassword)
        );

        eeprom_update_byte(&ee_magic, EEPROM_MAGIC
    );
        } else {
        eeprom_read_block(
            storedPassword,
            ee_password,
            sizeof(storedPassword)
        );
    }
}

void eeprom_save_password(const char *pwd) {
    char buf[MAX_PASSWORD_LENGTH + 1];
    strncpy(buf, pwd, MAX_PASSWORD_LENGTH);
    buf[MAX_PASSWORD_LENGTH] = '\0';
    eeprom_update_block(buf, ee_password, sizeof(
    buf));
}
```

Listing 18. Escritura y lectura de EEPROM

Logica de estados

```
typedef enum { UI_MENU, UI_INGRESO,
    UI_CAMBIO_ACTUAL, UI_CAMBIO_NUEVA, UI_ABIERTO,
     UI_ALARMA } ui_state_t;

int main(void){
    while(1){
        char key = keypad_scan();
        if (key) {
            switch (ui_state)
            {
                case UI_MENU: break;
                case UI_INGRESO: break;
                case UI_CAMBIO_ACTUAL: break;
                case UI_CAMBIO_NUEVA: break;
                case UI_CAMBIO_ALARMA: break;
                case UI_CAMBIO_ABIERTO: break;
    }
}
```

Listing 19. Logica de estados