Universidad de Concepción

Universidad de Concepción

Departamento de Ingeniería Eléctrica
UNIVERSIDAD DE CONCEPCIÓN

Taller de Sistemas digitales:Laboratorio 3

— Profesor Mario Medina —

Bruno Pacheco
Levi Sojos
Tomás Aguayo

Concepción, Chile

2025

Índice

1

Introducción

En el uso de sistemas digitales y los microcontroladores, el manejo de periféricos externos constituye una habilidad fundamental para el diseño de aplicaciones interactivas. El presente laboratorio tiene como objetivo implementar un sistema que integre distintos dispositivos de entrada y salida, específicamente un joystick, una matriz LED de 8x8 y una pantalla LCD1602 con interfaz I2C.

La actividad consiste en diseñar un circuito controlador que permita gestionar la matriz LED utilizando registros de desplazamiento. A través del joystick, el usuario puede interactuar con un juego en el que debe mover un "cazador" hasta un objetivo, evitando obstáculos y limitaciones de la matriz. Paralelamente, la pantalla LCD se emplea para mostrar mensajes de estado, tiempos de juego y resultados obtenidos, brindando un sistema completo de retroalimentación visual.

Este laboratorio busca reforzar los conocimientos adquiridos en experiencias previas, como el control de una matriz LED mediante el integrado MAX7219, pero ahora incorporando nuevas estrategias de control y comunicación. Además, fomenta la comprensión práctica de conceptos como multiplexación, registros de desplazamiento, comunicación I2C y conversión analógica-digital, los cuales son esenciales en el desarrollo de sistemas electrónicos modernos.

2

Marco teórico

El presente laboratorio aborda el uso de periféricos y técnicas de control digital aplicadas en sistemas embebidos, específicamente el manejo de un joystick, una matriz de LEDs 8x8 y un módulo LCD1602 con interfaz I2C. Estos elementos permiten implementar aplicaciones interactivas que combinan entrada y salida de datos, reforzando conceptos de electrónica digital y programación en microcontroladores.

2.1 Arduino Uno

El Arduino Uno es una placa de desarrollo basada en el microcontrolador ATmega328P de 8 bits, este opera a una frecuencia de 16 MHz. Posee 1 kB de memoria EEPROM, 2 kB de memoria SRAM y 32 kB de memoria flash [**medina-arduino**]. Este microcontrolador es ampliamente utilizada en proyectos educativos, de prototipado y aplicaciones de electrónica digital.

La placa cuenta con 14 pines digitales de entrada/salida (de los cuales 6 pueden usarse como salidas PWM), 6 entradas analógicas utilizando un conversor análogo-digital de 10 bits de resolución, un puerto USB para comunicación y alimentación, además de un regulador de voltaje que permite conectarla a fuentes externas.

Una de las características principales del Arduino Uno es su capacidad para interactuar con el entorno físico, permitiendo leer señales analógicas o digitales provenientes de sensores, procesarlas en el microcontrolador y generar respuestas mediante actuadores, como motores o LEDs.

2.2 Matriz LED 8x8

3 Diseño

3.1 Planteamiento de circuito

3.2 Desarrollo del código

El código se estructuró para gestionar simultáneamente las entradas del joystick, la lógica del juego (movimiento y colisiones), y el refresco de las salidas (matriz LED y LCD). El principio fundamental de diseño fue descargar el proceso de multiplexado de la matriz del ciclo principal (`loop()`) a las interrupciones por tiempo, permitiendo que el `loop()` se dedicara exclusivamente a la lógica del juego.

1. Optimización de Memoria con Campos de Bits

Para gestionar la posición del cazador (`player`) y el blanco (`goal`), se utilizó una estructura de datos `pos` que emplea **campos de bits** (bit fields):

Dado que la matriz es de 8×8 LEDs, las coordenadas X e Y solo necesitan valores de 0 a 7, lo que se puede representar con 3 bits ($2^3 = 8$). Esta técnica permitió empaquetar ambas coordenadas en un solo byte (6 bits utilizados), logrando una gestión de memoria muy eficiente.

El fragmento de código relevante es:

```
// Posición del cazador y del blanco (usando 3 bits por coordenada para
↪  ahorrar memoria)
struct pos {
  byte x : 3;
  byte y : 3;
} player, goal;
```

2. Estructura de la Matriz y Manipulación Bitwise

El estado visual del juego se mantiene en el arreglo de 8 bytes `matrix[8]`, donde cada byte representa el estado de una columna de la matriz LED. Esta representación es clave para la eficiencia de las actualizaciones.

- **Estructura Lógica:** `matrix[Y]` almacena los 8 LEDs de la columna Y. Un bit encendido (`1`) significa que el LED en esa posición debe estar activo.
- **Operaciones de Bits:** Las actualizaciones de posición (`player`) y el parpadeo (`goal`) se realizan mediante operaciones a nivel de bit. El índice de la matriz se ajusta dinámicamente utilizando el operador ternario (`? :`) para compensar la configuración física del cableado de la matriz.
  - **Encender/Actualizar (OR):** Se utiliza la operación **OR** para establecer el bit de la nueva posición del cazador en 1.
  - **Borrar/Parpadear (XOR):** Se utiliza la operación **XOR** para invertir el estado del bit. Esto se usa tanto para el parpadeo del blanco como para borrar la posición anterior del cazador.

```
// Cada 250ms leer entrada y mover al cazador
// También se hace parpadear al blanco
if (millis() - poll_time >= 250) {
  poll_time = millis();

  val_x = analogRead(JOY_X);
  val_y = analogRead(JOY_Y);
```

```
8
9   // Borrar punto de cazador anterior.
10  matrix[player.y == 0 ? 7 : (player.y - 1)] ^= (1 << player.x);
11
12   // Zona muerta de 256, o 50% de la resolución
13      // Sólo se actualiza la posición si no se va a chocar con la pared o el
        ↪  borde
14      if (val_x < 256 && (player.x > 0 && !(player.x == 5 && (player.y == 3 ||
        ↪  player.y == 4)))) player.x -= 1;
15      else if (val_x > 768 && (player.x < 7 && !(player.x == 2 && (player.y ==
        ↪  3 || player.y == 4)))) player.x += 1;
16      if (val_y < 256 && (player.y < 7 && !(player.y == 2 && (player.x == 3 ||
        ↪  player.x == 4)))) player.y += 1;
17      else if (val_y > 758 && (player.y > 0 && !(player.y == 5 && (player.x ==
        ↪  3 || player.x == 4)))) player.y -= 1;
18
19
20  // Actualizar bits en la matriz con la nueva posición
21  matrix[player.y == 0 ? 7 : (player.y - 1)] |= (1 << player.x);
22
23  // Parpadeo del blanco (cambiar su estado con XOR)
24  matrix[goal.y == 0 ? 7 : (goal.y - 1)] ^= (1 << goal.x);
25 }
```

3. Control de Refresco mediante Interrupciones (Timer1)

El refresco de la matriz LED, crucial para evitar el efecto de parpadeo, se implementó usando el Timer1 del microcontrolador. Esto garantiza que la actualización sea periódica y no dependa del tiempo de ejecución del `loop()`. Se configuraron dos interrupciones (ISRs):

- **`ISR(TIMER1_COMPA_vect)`**: Se ejecuta a 2500 Hz. Es la responsable de **desactivar** la fila actual para prevenir ela persistencia de la imagen anterior, e inmediatamente carga los datos de columna (`DCOL`) para la *siguiente* fila a dibujar.
- **`ISR(TIMER1_COMPB_vect)`**: Se ejecuta a 5000 Hz, actuando 200 μs después de la interrupción A. Su función es **activar** la fila correspondiente al `counter` actual, encendiendo así la fila con los datos de columna previamente cargados.

Esta técnica asegura un ciclo de trabajo constante y una frecuencia de refresco elevada (aproximadamente 125Hz por fila) para evitar el parpadeo perceptible.

La configuración del temporizador en `setup()` y los ISRs son:

```
1  // -- Configuración de interrupts de tiempo (Timer1 para multiplexado de
   ↪  matriz) --
2    TCCR1A = 0; // Limpiar registros A
3    TCCR1B = 0; // Limpiar registros B
4    // ... (otros registros de configuración de Timer1)
5    TIMSK1 |= (1 << OCIE1A) | (1 << OCIE1B);
```

```
  OCR1A = 6400; // Valor de comparación A (400us)
  OCR1B = 3200; // Valor de comparación B (200us)
// ----------------------------------------
ISR(TIMER1_COMPA_vect) {
  // 1. Deshabilitar la fila actual (blanquear) para evitar ghosting
  digitalWrite(LROW, LOW);
  shiftOut(DROW, CLK1, LSBFIRST, 0x00);
  // ...
}
ISR(TIMER1_COMPB_vect) {
  // 1. Activar la fila 'counter'
  digitalWrite(LROW, LOW);
  shiftOut(DROW, CLK1, LSBFIRST, (1 << counter));
  digitalWrite(LROW, HIGH);
}
```

4. Lógica del Juego y Detección de Colisión La función `loop()` se encarga de la lógica principal:

1. **Polling del Joystick:** La lectura de las entradas analógicas (`JOY_X`, `JOY_Y`) se realiza cada 250ms para un movimiento controlado, estableciendo una zona muerta para evitar movimientos involuntarios.
2. **Actualización y Colisión:** Se calcula la nueva posición del jugador, aplicando reglas de detección de colisión que impiden que el cazador se mueva sobre las coordenadas fijas del muro ($x = 3, 4$ y $y = 3, 4$). La posición se actualiza en el arreglo `matrix` utilizando operaciones de bits (`OR` para encender, `XOR` para borrar la posición anterior).
3. **Fin de Partida:** Se evalúan dos condiciones de término: victoria (cuando `player` y `goal` coinciden) y derrota por tiempo límite (5s), actualizando el estado del juego (`game_state`) y la pantalla LCD.

El control de movimiento en `loop()` incluye una lógica de colisión para evitar que el cazador se mueva a las posiciones ocupadas por el muro estático. Por ejemplo, la prevención de movimiento a la derecha (`player.x += 1`) cuando se está justo a la izquierda del muro:

```
        // Choque a la DERECHA: player.x = 2, y el movimiento es a la derecha
        ↪  (hacia columna 3 o 4)
else if (val_x > 768 && player.x < 7) {
  if (!((player.x == 2) && (player.y == 3 || player.y == 4))) player.x += 1;
}
```

4

Implementación

5

Conclusión

A

Códigos