Integrantes:

Steven Cisneros

Ricardo Vilcacundo

Daniel Bejarano

Paralelo:

P2

Docente:

Ronald Solis

Nombre del Proyecto:

Minijuego Laberinto

TAREA DE SISTEMAS EMBEBIDOS - TEÓRICO

TAREA #3
Grupal

COMUNICACIÓN ENTRE MICROCONTROLADORES ATMEGA328P - PIC16F887

1. Objetivo General

- Desarrollar un sistema de juego interactivo de 3 niveles de dificultad que combine salidas visuales mediante una matriz LED 8x8 controlada por el microcontrolador ATmega328P, y salidas auditivas a través de un sistema de reproducción de melodías con el microcontrolador PIC16F887, utilizando entradas mediante pulsadores e implementando comunicación entre ambos dispositivos, para reforzar habilidades en programación y control de sistemas embebidos, simulando su funcionamiento completo en Proteus. El sistema deberá ser programado en C para ambos microcontroladores.

2. Objetivos Específicos

- Diseñar e implementar un sistema de control de una matriz LED 8x8 con el microcontrolador ATmega328P para desplegar caracteres, símbolos y animaciones del juego.
- Programar un sistema de reproducción de melodías utilizando el PIC16F887, capaz de interpretar comandos recibidos desde el ATmega328P para vincular los sonidos a los eventos del juego.
- Integrar botones físicos como entradas al ATmega328P para interactuar con el juego, controlando su flujo (iniciar, seleccionar, reiniciar, etc.) y permitiendo que el sistema actúe según la interacción del usuario.
- Implementar 3 niveles de dificultad (fácil, medio, difícil) que modifiquen dinámicamente parámetros del juego como la velocidad, lógica o complejidad, y visualizar brevemente el nivel actual en la matriz LED al cambiarlo.
- Establecer un canal de comunicación paralelo entre el ATmega328P y el PIC16F887 que permita la sincronización entre efectos visuales y sonoros.
- Simular de manera completa el juego en el entorno Proteus, integrando los dos microcontroladores, la matriz LED y el sistema de audio.
- Documentar el proceso de diseño, desarrollo y simulación en un informe técnico, incluyendo un repositorio GitHub con el código fuente y evidencias.

3. Tarea

Cada grupo deberá desarrollar un sistema de juego interactivo programado en lenguaje C con 3 niveles de dificultad (fácil, medio, difícil) que cambien dinámicamente algún parámetro del juego (velocidad de animaciones, número de intentos, lógica de botones, etc.), donde el ATmega328P controlará una matriz LED 8x8 para mostrar los símbolos y dinámicas del juego; además de utilizar pulsadores como entradas que permitan al usuario interactuar con el juego, mientras que el PIC16F887 será responsable de reproducir melodías relacionadas con los eventos del juego (inicio, victoria, error, etc.). Ambos microcontroladores deben estar conectados mediante comunicación serial para intercambiar datos. El sistema deberá ser simulado y validado completamente en Proteus.

Juego Desarrollado

El juego desarrollado consiste en un laberinto con obstáculos de 3 niveles. Estos fueron hechos usando como plantilla la matriz de LEDs de 8x8. A continuación, se presentan los 3 niveles:

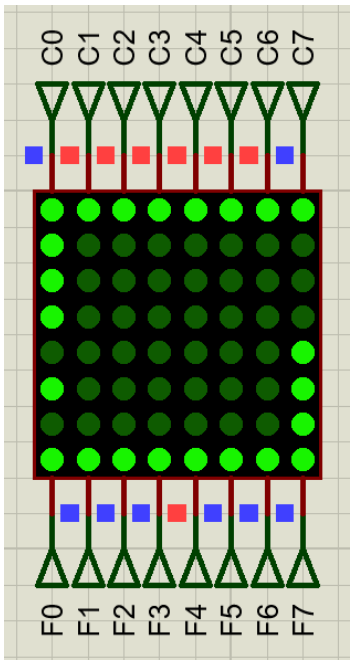


Ilustración 1. Nivel Fácil

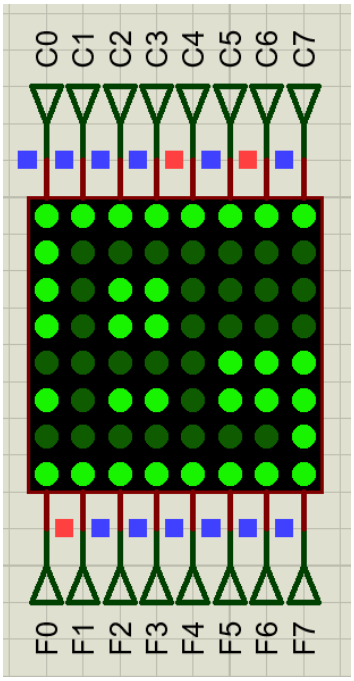


Ilustración 2. Nivel Intermedio

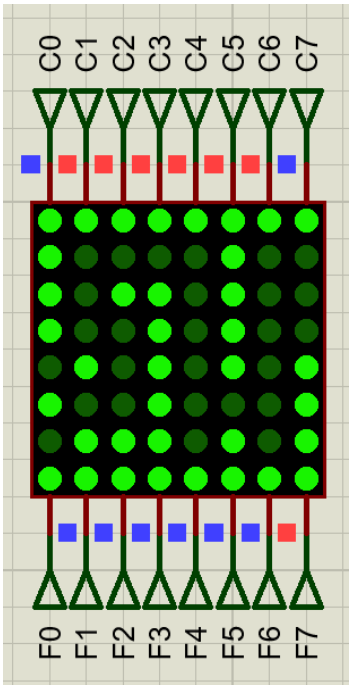


Ilustración 3. Nivel Difícil

Como podemos observar, existe un aumento gradual de la dificultad en los niveles del juego. El desplazamiento del personaje se realiza mediante la pulsación de botones que posteriormente son procesados por el ATMEGA328P. En la siguiente imagen se puede observar la disposición de dichos botones.

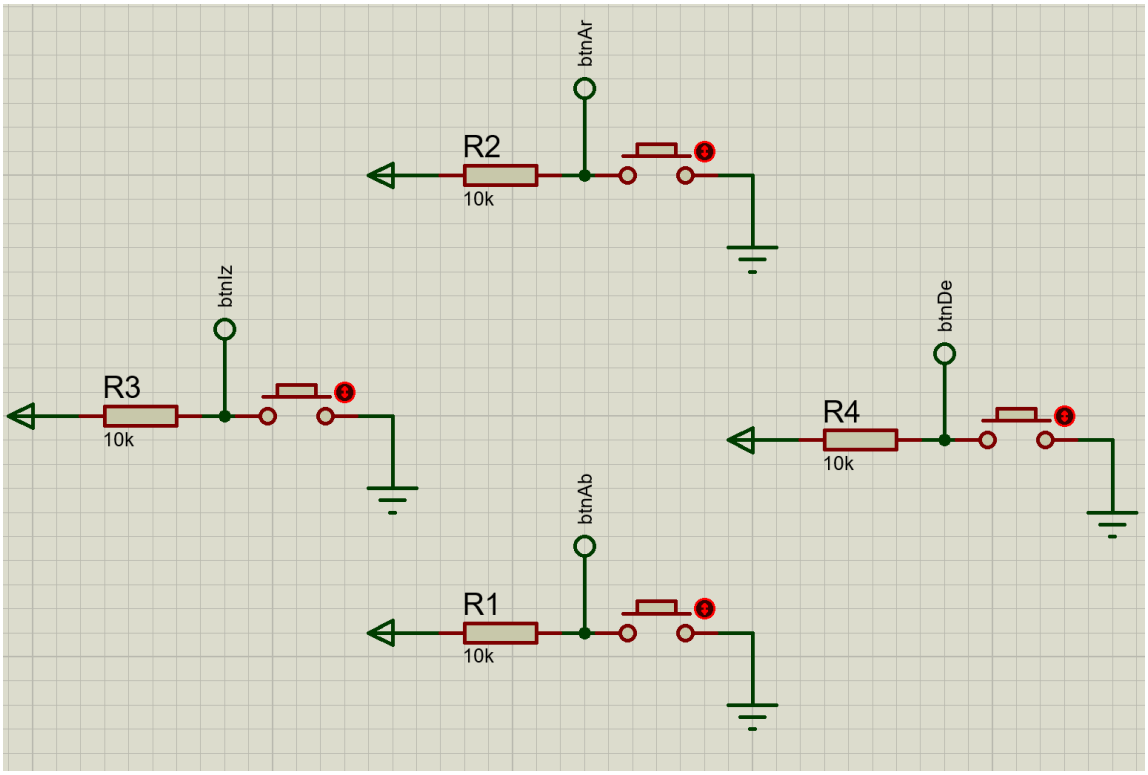


Ilustración 4. Botones de Desplazamiento

Además, existe una conexión entre el ATMEGA328P y el PIC16F887 la cual habilita la reproducción de audios a través de un Speaker.

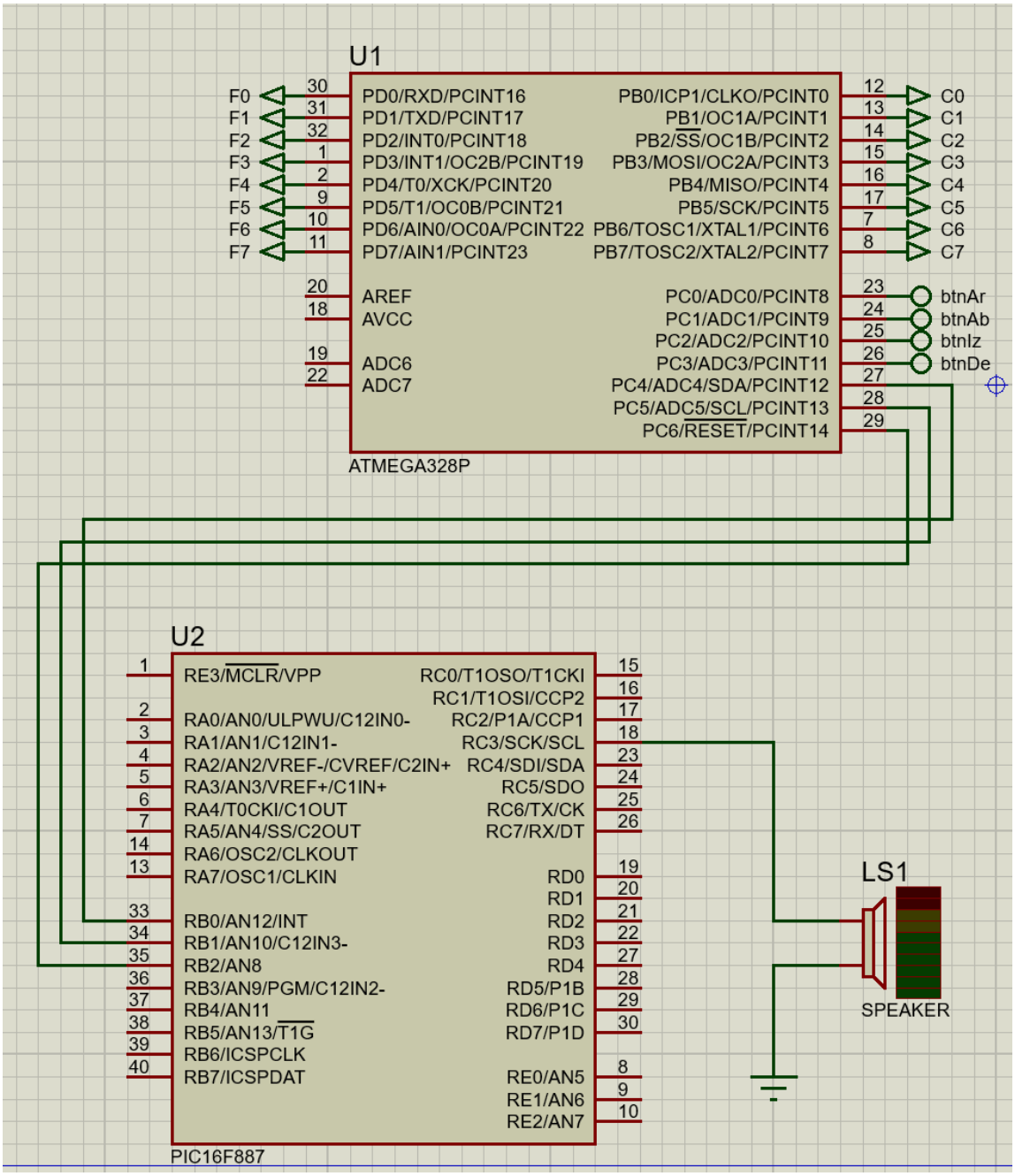
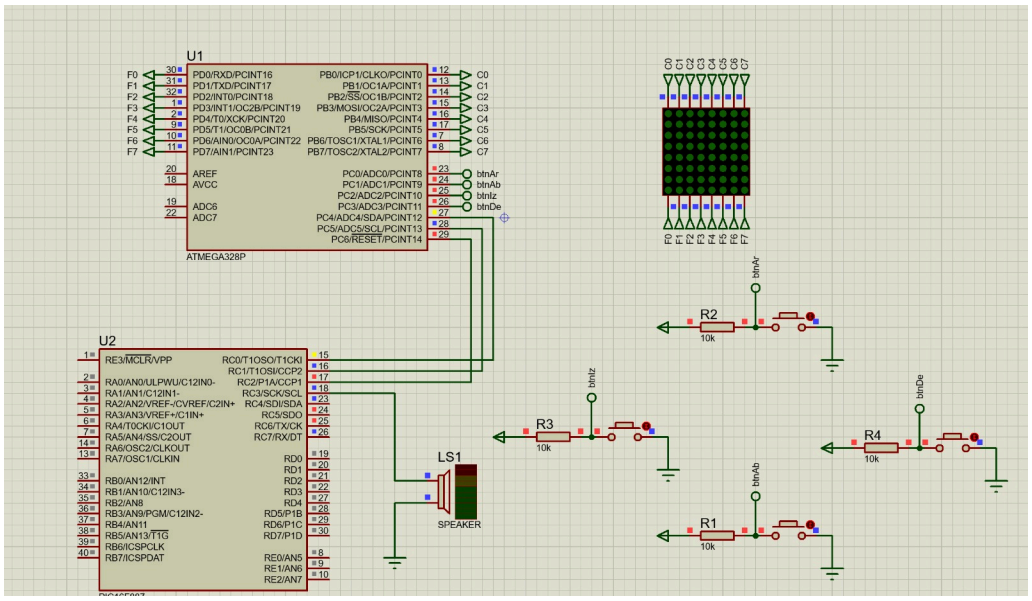
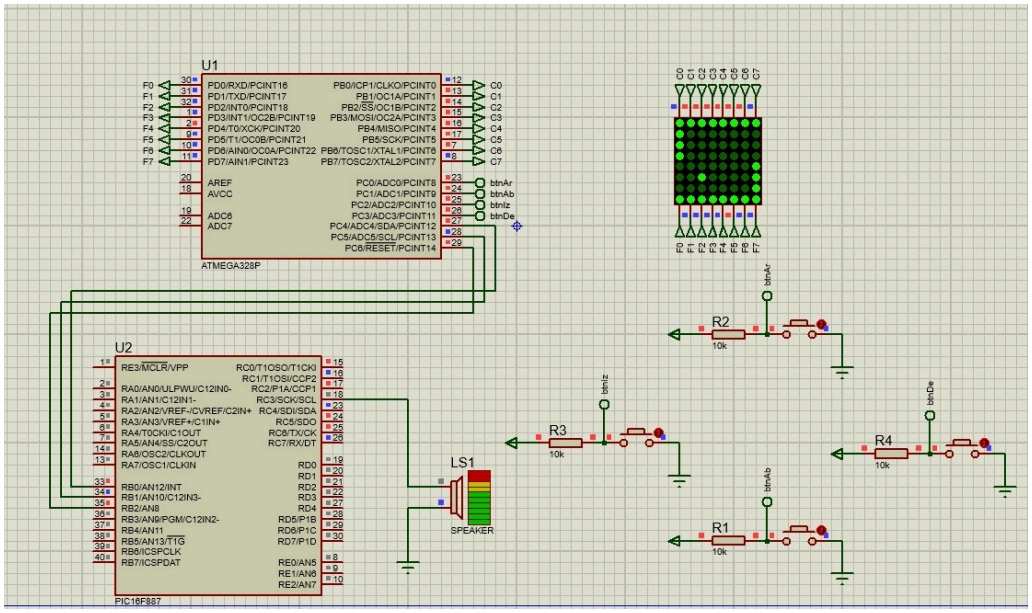
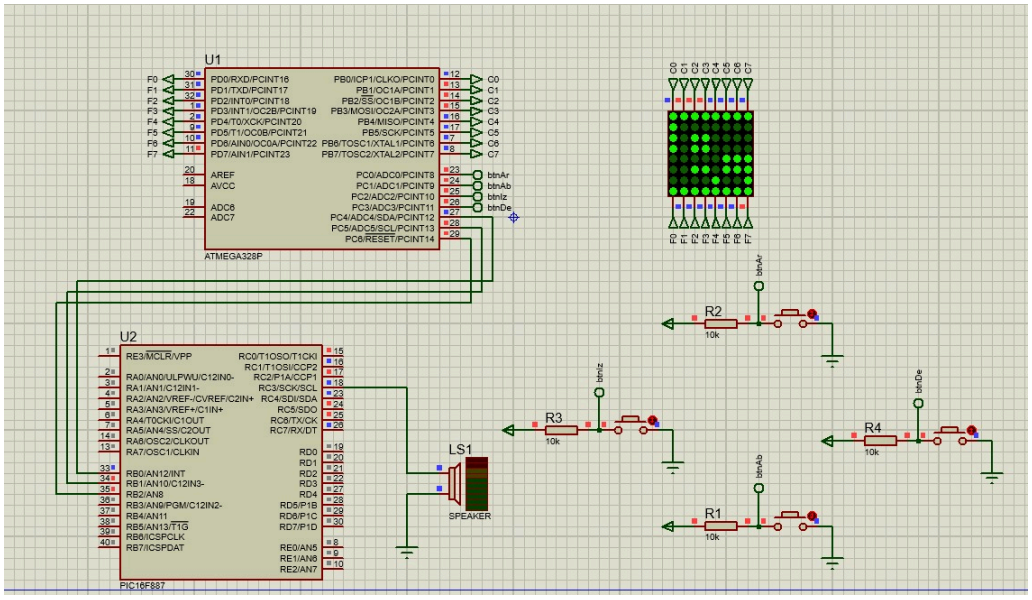


Ilustración 5. Conexión Atmega328P con PIC16F887 y su Speaker

4. Capturas de simulación en Proteus







5. Explicación del código para cada microcontrolador Atmega328p

```c
#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

// Botones
#define BTN_IZQ PC0
#define BTN_DER PC1
#define BTN_ARR PC2
#define BTN_ABA PC3

// LEDs de estado
#define PIN_ENTRADA PD6
#define PIN_META    PD7

// Jugador
uint8_t jugador_x = 5;
uint8_t jugador_y = 0;

// Meta final
uint8_t meta_x = 2;
uint8_t meta_y = 7;

// Nivel actual y timer
uint8_t nivel_actual = 1;
uint16_t tiempo_restante = 0;

// Laberintos
const uint8_t laberinto_nivel1[8] = {
    0b10001111, 0b10000001, 0b10000001, 0b10000001,
    0b10000001, 0b10000001, 0b10000001, 0b11110001
};
const uint8_t laberinto_nivel2[8] = {
    0b10001111, 0b10000001, 0b10101101, 0b10101101,
    0b10000001, 0b10110001, 0b10110001, 0b11110001
};
const uint8_t laberinto_nivel3[8] = {
    0b10001111, 0b11010001, 0b11000101, 0b11111101,
    0b10000001, 0b11011111, 0b10000001, 0b11110001
};

// Mapeo de filas
const uint8_t filas[8] = { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };

void init_ports() {
    DDRD = 0xFF;
    DDRB = 0xFF;
    DDRC &= ~((1<<BTN_IZQ)|(1<<BTN_DER)|(1<<BTN_ARR)|(1<<BTN_ABA));
    PORTC |= (1<<BTN_IZQ)|(1<<BTN_DER)|(1<<BTN_ARR)|(1<<BTN_ABA);
}

const uint8_t* obtener_laberinto() {
    switch (nivel_actual) {
        case 1: return laberinto_nivel1;
        case 2: return laberinto_nivel2;
        case 3: return laberinto_nivel3;
        default: return laberinto_nivel1;
    }
}

uint8_t es_pared(uint8_t x, uint8_t y) {
    return obtener_laberinto()[y] & (1 << x);
}

void mostrar_laberinto(uint8_t incluir_jugador) {
    for (uint8_t fila = 0; fila < 8; fila++) {
        PORTD = filas[fila];

        uint8_t salida = obtener_laberinto()[fila];

        if (incluir_jugador && fila == jugador_y) salida |= (1 << jugador_x);
        if (fila == meta_y) salida &= ~(1 << meta_x); // puerta salida

        PORTB = ~salida;
        _delay_ms(0.5);
    }
    PORTD = 0;
}
```

```c
//animacion de victoria
void mostrar_victoria() {
    for (uint8_t i = 0; i < 10; i++) {
        PORTB = 0x00;
        PORTD = 0xFF;
        _delay_ms(20);
        PORTD = 0x00;
        _delay_ms(20);
    }
}

//animacion de derrota
void mostrar_derrota() {
    for (uint8_t i = 0; i < 5; i++) {
        PORTD = 0b10101010;
        PORTB = 0b01010101;
        _delay_ms(100);
        PORTD = 0x00;
        PORTB = 0xFF;
        _delay_ms(100);
    }
}

void mover_jugador() {
    static uint8_t last_state = 0;
    uint8_t actual_state = PINC;

    if (!(actual_state & (1 << BTN_IZQ)) && (last_state & (1 << BTN_IZQ))) {
        if (jugador_x > 0 && !es_pared(jugador_x - 1, jugador_y)) jugador_x--;
    }

    if (!(actual_state & (1 << BTN_DER)) && (last_state & (1 << BTN_DER))) {
        if (jugador_x < 7 && !es_pared(jugador_x + 1, jugador_y)) jugador_x++;
    }

    if (!(actual_state & (1 << BTN_ARR)) && (last_state & (1 << BTN_ARR))) {
        if (jugador_y > 0 && !es_pared(jugador_x, jugador_y - 1)) jugador_y--;
    }

    if (!(actual_state & (1 << BTN_ABA)) && (last_state & (1 << BTN_ABA))) {
        if (jugador_y < 7 && !es_pared(jugador_x, jugador_y + 1)) jugador_y++;
    }

    last_state = actual_state;
}

void actualizar_led_estado() {
    if (jugador_x == 1 && jugador_y == 1) {
        PORTD |= (1 << PIN_ENTRADA);
    } else {
        PORTD &= ~(1 << PIN_ENTRADA);
    }

    if (jugador_x == meta_x && jugador_y == meta_y) {
        PORTD |= (1 << PIN_META);
    } else {
        PORTD &= ~(1 << PIN_META);
    }
}

void siguiente_nivel() {
    mostrar_victoria();
    nivel_actual++;
    jugador_x = 5;
    jugador_y = 0;

    if (nivel_actual == 3) {
        tiempo_restante = 180;
        meta_x = 2;
        meta_y = 7;
    }
}
```

```c
int main(void) {
    init_ports();

    while (1) {
        mover_jugador();
        actualizar_led_estado();

        for (uint8_t i = 0; i < 5; i++) {
            mostrar_laberinto(1);
        }

        if (nivel_actual == 3) {
            if (tiempo_restante > 0) {
                tiempo_restante--;
            } else {
                mostrar_derrota();
                nivel_actual = 1;
                jugador_x = 5;
                jugador_y = 0;
                continue;
            }
        }

        if (jugador_x == meta_x && jugador_y == meta_y) {
            if (nivel_actual == 3) {
                mostrar_victoria();
                while (1); // fin del juego
            } else {
                siguiente_nivel();
            }
        }

        _delay_ms(0.5);
    }
}
```

Configuración del CPU y Librerías:

#define F_CPU 8000000UL: Define la frecuencia de operación del microcontrolador en 8 MHz, esencial para cálculos de tiempo precisos con _delay_ms().

#include <avr/io.h>: Proporciona las definiciones de los registros de E/S del ATmega328P.

#include <util/delay.h>: Permite el uso de funciones de retardo temporizado.

Definiciones de Hardware:

Botones: BTN_IZQ, BTN_DER, BTN_ARR, BTN_ABA se mapean a los pines PC0 a PC3 del Puerto C.

LEDs de Estado: PIN_ENTRADA (PD6) y PIN_META (PD7) se usan para indicar puntos específicos en el laberinto.

Variables de Estado del Juego:

jugador_x, jugador_y: Posición actual del jugador en la cuadrícula 8x8.

meta_x, meta_y: Coordenadas de la salida del laberinto.

nivel_actual: Controla el nivel del juego.

tiempo_restante: Usado para un temporizador en el nivel 3.

Laberintos: Se definen tres arrays laberinto_nivel1, laberinto_nivel2, laberinto_nivel3. Cada elemento del array representa una fila de la matriz 8x8, donde los bits '1' son paredes y '0' son caminos (aunque la visualización en PORTB = ~salida; invierte esto).

Mapeo de Filas (filas): Un array que facilita la selección individual de cada fila de la matriz de LEDs para la multiplexación.

init_ports():

Configura los Puertos D y B como salidas (DDRD = 0xFF; DDRB = 0xFF;) para controlar la matriz de LEDs.

Configura los pines de los botones en el Puerto C como entradas y habilita sus resistencias pull-up internas (PORTC |= ...).

Lógica del Laberinto:

obtener_laberinto(): Devuelve el laberinto correspondiente al nivel actual.

es_pared(x, y): Verifica si una celda específica del laberinto es una pared, impidiendo el paso del jugador.

Control de Pantalla (mostrar_laberinto()):

Implementa la técnica de multiplexación para refrescar la matriz de LEDs 8x8.

Activa una fila a la vez (usando PORTD) y luego envía el patrón de LEDs para las columnas de esa fila (usando PORTB, invirtiendo los bits).

Añade el jugador y la "puerta" de la meta al patrón de la fila si corresponde.

_delay_ms(0.5): Pequeño retardo crucial para la persistencia de la visión.

Animaciones:

mostrar_victoria() y mostrar_derrota(): Proporcionan efectos visuales sencillos en la matriz de LEDs para indicar el éxito o fracaso.

Movimiento del Jugador (mover_jugador()):

Lee el estado de los botones y actualiza la posición del jugador_x, jugador_y verificando los límites de la pantalla y la presencia de paredes.

Utiliza una variable last_state para detectar solo el *flanco de bajada* de las pulsaciones de botón, evitando movimientos múltiples si el botón se mantiene presionado.

Actualización de LEDs de Estado (actualizar_led_estado()):

Controla los LEDs PIN_ENTRADA y PIN_META para proporcionar retroalimentación visual sobre la posición del jugador en relación con la entrada y la meta del laberinto.

Gestión de Nivel (siguiente_nivel()):

Avanza al siguiente nivel del juego, reinicia la posición del jugador y, para el nivel 3, inicializa un contador de tiempo.

Bucle Principal (main()):

init_ports(): Inicializa todos los puertos al inicio.

while (1): El bucle infinito donde se ejecuta la lógica del juego.

Procesa el movimiento del jugador (mover_jugador()).

Actualiza los LEDs de estado (actualizar_led_estado()).

Refresca la matriz de LEDs (mostrar_laberinto()) varias veces para asegurar una visualización constante y sin parpadeos perceptibles.

Gestiona el temporizador para el nivel 3, mostrando la derrota si el tiempo se agota.

Comprueba si el jugador ha alcanzado la meta, avanzando de nivel o finalizando el juego.

_delay_ms(0.5): Un pequeño retardo para controlar el ritmo del bucle principal.

PIC16F887

```
// -------------------- DECLARACIÓN DE FUNCIONES --------------------
void DelayNote() { Delay_ms(50); }

void Tone(unsigned int freq, unsigned int duration) {
   Sound_Play(freq, duration);
   DelayNote();
}

// ---------- MELODÍA 1: INICIO (Super Mario Bros adaptado) ----------
void Melody_Intro() {
   Tone(659, 200); Tone(659, 200); Tone(659, 200);
   Tone(523, 200); Tone(659, 200); Tone(784, 200);
   Tone(392, 400);
}

// ---------- MELODÍA 2: GANASTE NIVEL (Zelda) ----------
void Melody_WinLevel() {
   Tone(659, 150); Tone(784, 150); Tone(659, 150);
   Tone(784, 150); Tone(659, 150); Tone(880, 300);
}

// ---------- MELODÍA 3: DERROTA (Pac-Man muerte) ----------
void Melody_Lose() {
   Tone(880, 150); Tone(830, 150); Tone(784, 150);
   Tone(740, 150); Tone(698, 300); Tone(659, 300);
}

// ---------- MELODÍA 4: JUEGO COMPLETADO (Final Fantasy Victory) ----------
void Melody_WinGame() {
   Tone(523, 150); Tone(659, 150); Tone(784, 150); Tone(1046, 300);
   Tone(784, 150); Tone(880, 150); Tone(988, 300); Tone(1046, 300);
}

// ---------- MELODÍA INICIO: Tetris Type A ----------
void Melody_Intro_Tetris() {
   Tone(659, 150); Tone(494, 150); Tone(523, 150); Tone(587, 150);
   Tone(523, 100); Tone(494, 100); Tone(440, 150);
   Tone(440, 100); Tone(523, 100); Tone(659, 150);
   Tone(587, 100); Tone(523, 100); Tone(494, 200);
}

// ---------- MELODÍA VICTORIA: Sonic Stage Clear ----------
void Melody_WinLevel_Sonic() {
   Tone(659, 150); Tone(784, 150); Tone(880, 200);
   Tone(784, 150); Tone(659, 300);
}

// ---------- MELODÍA DERROTA: Donkey Kong Game Over ----------
void Melody_Lose_DonkeyKong() {
   Tone(440, 300); Tone(392, 300);
   Tone(370, 300); Tone(349, 400);
}

// ---------- MELODÍA JUEGO COMPLETADO: Pokémon Level Up ----------
void Melody_WinGame_Pokemon() {
   Tone(659, 100); Tone(698, 100); Tone(784, 200);
   Tone(880, 150); Tone(988, 300);
}


// -------------------- PROGRAMA PRINCIPAL --------------------
void main() {
   // Configuración de puertos
   ANSEL = 0; ANSELH = 0;        // Todos los pines como digitales
   C1ON_bit = 0; C2ON_bit = 0;   // Comparadores OFF
   TRISC = 0x00;                 // RC3 como salida
   Sound_Init(&PORTC, 3);        // Inicializar sonido en RC3

   while (1) {
         Melody_Intro();
         Delay_ms(2000);
         Melody_WinLevel();
         Delay_ms(2000);
         Melody_Lose();
         Delay_ms(2000);
         Melody_WinGame();
         Delay_ms(2000);
   }
}
```

Librería de Sonido: El código asume la existencia de una librería Sound_Play(freq, duration) y Delay_ms(), que son funciones comunes en entornos de desarrollo para PIC como MikroC.

Sound_Play(frecuencia, duración): Esta función tomará una frecuencia (en Hertz) y una duración (en milisegundos) para generar un tono en un pin específico.

Delay_ms(milisegundos): Esta función introduce un retardo en la ejecución del programa.

Funciones Auxiliares de Audio:

void DelayNote(): Introduce un pequeño retardo de 50 ms después de cada nota. Esto ayuda a separar las notas en una melodía, haciendo que suenen más distintivas.

void Tone(unsigned int freq, unsigned int duration): Es una envoltura de Sound_Play. Facilita la reproducción de una nota musical con una frecuencia y duración dadas, seguida por el DelayNote().

Definición de Melodías: El código incluye varias funciones que definen melodías específicas para diferentes eventos del juego. Cada melodía es una secuencia de llamadas a Tone(), con frecuencias y duraciones ajustadas para replicar temas de videojuegos conocidos:

Melody_Intro(): Inspirada en Super Mario Bros.

Melody_WinLevel(): Inspirada en Zelda.

Melody_Lose(): Inspirada en la muerte de Pac-Man.

Melody_WinGame(): Inspirada en la victoria de Final Fantasy.

Melody_Intro_Tetris(): Tema A de Tetris.

Melody_WinLevel_Sonic(): Limpiar etapa de Sonic.

Melody_Lose_DonkeyKong(): Game Over de Donkey Kong.

Melody_WinGame_Pokemon(): Subida de nivel en Pokémon.

Configuración de Puertos (main()):

ANSEL = 0; ANSELH = 0;: Configura todos los pines como digitales, desactivando sus funciones analógicas.

C1ON_bit = 0; C2ON_bit = 0;: Desactiva los comparadores analógicos.

TRISC = 0x00;: Configura todo el Puerto C como salida. Esto es fundamental porque el pin utilizado para generar sonido (RC3) debe ser una salida.

Sound_Init(&PORTC, 3);: Inicializa el módulo de generación de sonido, indicando que el sonido se producirá a través del pin RC3 del Puerto C.

6. Explicación del esquema de comunicación entre microcontroladores

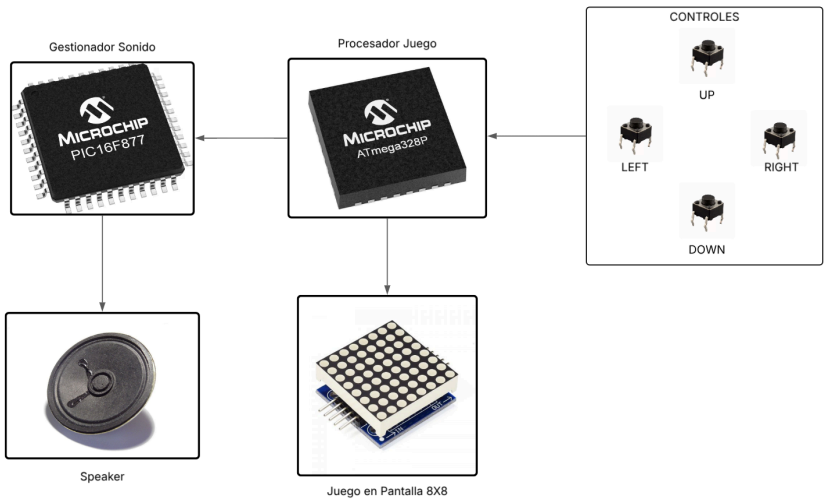


Ilustración 6. Esquema del Sistema de Juego

El esquema de comunicación representa la arquitectura de un videojuego de laberinto, donde las funcionalidades se distribuyen para optimizar el rendimiento. El corazón de este sistema, reside en el Procesador de Juego (Microchip ATmega328P), que gestiona la lógica central del juego, la interacción del jugador y la visualización. El Gestionador de Sonido (Microchip PIC16F877) y el Speaker son componentes dedicados a la experiencia auditiva.

La comunicación entre el ATmega328P y el PIC16F877 sigue un modelo de emisor-receptor, donde el ATmega328P asume el rol de emisor de comandos y el PIC16F877 actúa como receptor y ejecutor de acciones de audio.

El "Procesador de Juego", es el encargado de determinar cuándo y qué tipo de sonido debe reproducirse. Para ello, utiliza tres de sus pines de propósito general: PC4, PC5 y PC6. Estos pines son configurados como salidas y se activan o desactivan en patrones específicos para codificar la información del sonido deseado.

El "Gestionador de Sonido", tiene sus pines RB0, RB1 y RB2 configurados como entradas. Estos pines están directamente conectados a PC4, PC5 y PC6 del ATmega328P, respectivamente. El PIC16F877 monitorea constantemente el estado de estos pines.

El PIC16F877, al recibir y decodificar estos patrones en sus pines RB0, RB1 y RB2, busca la equivalencia interna en su propia programación. Una vez identificado el comando, el PIC16F877 procede a generar la señal de audio correspondiente. Esta generación se realiza internamente en el PIC, probablemente utilizando técnicas como Modulación por Ancho de Pulso (PWM) o bit-banging para producir diferentes tonos y efectos sonoros que son enviados finalmente al Speaker para su audición.

7. Conclusiones

- La arquitectura de dos microcontroladores (ATmega328P para lógica y visualización, PIC16F877 para sonido) demostró ser una estrategia efectiva para distribuir la carga de procesamiento. Esto facilita el desarrollo y depuración, ya que cada microcontrolador se especializa en un conjunto de tareas, optimizando el rendimiento general del sistema.

- Se logró implementar un sistema de control de matriz LED 8x8 con el ATmega328P, capaz de desplegar laberintos, la posición del jugador y animaciones, lo que es crucial para la interfaz visual del juego. La multiplexación por tiempo permite una visualización fluida a pesar de la limitación de pines.
- El PIC16F877 se configuró exitosamente para reproducir una variedad de melodías y efectos de sonido, liberando al microcontrolador principal de esta tarea y garantizando una experiencia auditiva complementaria a los eventos del juego.
- La integración de pulsadores al ATmega328P como entradas permitió una interacción intuitiva por parte del usuario, fundamental para el control del personaje en el laberinto y la navegación del juego.

8. Recomendaciones

- La comunicación actual mediante la lectura constante de pines (Polling) por parte del PIC consume ciclos de CPU innecesarios y puede llevar a la pérdida de información si el ATmega328P envía los comandos demasiado rápido o si el PIC está ocupado con la reproducción de una melodía larga. Se recomienda encarecidamente implementar la comunicación utilizando interrupciones externas en el PIC16F877. Esto permitiría que el PIC solo se active cuando el ATmega328P le envíe un nuevo comando de sonido, respondiendo de forma más eficiente y en tiempo real, sin necesidad de un bucle de espera constante.
- Si bien el envío de patrones de bits es funcional, para sistemas más complejos o con más comandos de sonido, se podría considerar un protocolo de comunicación serie (como UART o I2C) entre los microcontroladores. Esto permitiría enviar una mayor variedad de comandos y datos de forma más estructurada, facilitando la escalabilidad del sistema y la adición de nuevas funcionalidades de audio o retroalimentación entre los dispositivos.
- Para lograr una sincronización perfecta entre los efectos visuales de la matriz LED y los sonidos, especialmente en animaciones rápidas o eventos críticos del juego, se podría implementar un mecanismo de "handshake" o confirmación. El ATmega328P enviaría un comando y esperaría una confirmación del PIC antes de proceder con el siguiente paso visual, asegurando que el sonido se reproduzca en el momento preciso del evento en pantalla.

Enlaces

https://www.youtube.com/watch?v=2r0FJbt8_VA
https://github.com/RicardoVilcacundo/deber3SE