ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y COMPUTACIÓN

SISTEMAS EMBEBIDOS – TEÓRICO

COMUNICACIÓN ENTRE MICROCONTROLADORES

INTEGRANTES:

BAJAÑA VALAREZO JOSUÉ ROBERTO

ESTRADA SANCHEZ ROBERTO ALEJANDRO

PARALELO:

2

PROFESOR:

SOLIS MESA RONALD DAVID

AYUDANTE:

MUÑOZ BURGOS OSCAR TOMAS

FECHA Y PERIODO ACADÉMICO:

14 de noviembre del 2025

PAO II – 2025

1. INTRODUCCIÓN

En el campo de los sistemas embebidos, la capacidad de crear dispositivos interactivos que respondan en tiempo real a las entradas del usuario es fundamental. Estos sistemas a menudo requieren una gestión compleja de múltiples tareas simultáneas, como el procesamiento de la lógica de una aplicación, el manejo de interfaces de usuario visuales y la generación de retroalimentación auditiva. El diseño de tales sistemas presenta desafíos significativos en la integración de hardware y la sincronización de software, exigiendo soluciones robustas para garantizar un rendimiento fluido y coherente.

Para abordar esta complejidad, el presente proyecto implementa un sistema de juego interactivo mediante una arquitectura de procesamiento distribuido que utiliza dos microcontroladores distintos: un ATmega328P y un PIC16F887. En este diseño, el ATmega328P actúa como la unidad de control principal, gestionando la lógica del juego, la lectura de entradas del usuario a través de pulsadores y el control de una matriz LED 8x8 para el despliegue de la interfaz visual. Por su parte, el PIC16F887 opera como un procesador esclavo dedicado, responsable exclusivo de la reproducción de melodías y efectos de sonido.

El valor de este informe es documentar el diseño, desarrollo y validación de dicho sistema. Se detalla la implementación de tres niveles de dificultad y el establecimiento de un canal de comunicación entre ambos microcontroladores, un aspecto crítico para sincronizar los eventos visuales con la retroalimentación auditiva. Todo el sistema fue programado en lenguaje C y su funcionamiento integral fue comprobado mediante simulación en el entorno Proteus, validando así la viabilidad y correcta operación del diseño propuesto.

2. OBJETIVOS

2.1. OBJETIVO GENERAL

Construir un prototipo funcional del videojuego arcade "Stacker", mediante la integración de dos microcontroladores, un ATmega328P para la gestión de la lógica del juego, el control por multiplexación de una matriz LED 8x8 y la lectura de pulsadores, y un PIC16F887 dedicado a la generación de audio. Esto se logrará estableciendo un canal de comunicación de un solo hilo (basado en ancho de pulso) entre ambos CIs y validando el diseño completo en Proteus, con la finalidad de aplicar y consolidar los conocimientos en el diseño, programación y simulación de sistemas embebidos.

2.2. OBJETIVOS ESPECÍFICOS

Programar la gestión de una matriz LED 8x8, mediante el uso del microcontrolador ATmega328P, para desplegar las animaciones, símbolos y caracteres visuales que conforman la interfaz gráfica del juego interactivo.

Implementar un sistema de audio basado en eventos, mediante la programación del microcontrolador PIC16F887 para controlar un buzzer, con el propósito de reproducir melodías específicas (inicio, error, victoria) que brinden retroalimentación auditiva al usuario según su progreso en el juego.

Sincronizar los efectos visuales y auditivos del juego, mediante el establecimiento de un canal de comunicación entre el ATmega328P (maestro visual) y el PIC16F887 (esclavo de audio), para asegurar que las melodías se activen de forma coherente y simultánea a los eventos mostrados en la matriz LED.

3. DESARROLLO

3.1. INSTRUCCIONES

El desarrollo del proyecto se divide en fases clave, comenzando por el diseño del hardware en Proteus. Se construye el esquema integrando: el ATmega328P, el PIC16F887, una matriz LED 8x8 (controlada por multiplexación directa a los puertos B y D), dos pulsadores de entrada (conectados a PC0 y PC1), un buzzer (controlado por RD7 del PIC) y la conexión de comunicación de un solo hilo "SONIDO" (desde PC2 del ATmega a RB0 del PIC). Para la programación, se utiliza Visual Studio Code con PlatformIO para el ATmega328P y MikroC for PIC para el PIC16F887.

La implementación funcional se centra en el ATmega328P, que ejecuta toda la lógica del juego. Este gestiona el menú de selección de nivel, el bucle principal del juego (ejecutar_juego), el movimiento de los bloques y el cálculo de aciertos contra la máscara de la fila anterior.

El pulsador en PC0 se usa para "detener" el bloque, mientras que el de PC1 se usa para navegar el menú de nivel y reclamar el "premio chico" en la cuarta fila. El PIC16F887, por su parte, se programa en un bucle infinito que mide el ancho del pulso en su pin de entrada (medir_pulso_ms) y, según el rango de tiempo medido, reproduce la melodía correspondiente.

Finalmente, la lógica de la aplicación integra todos los elementos. La sincronización se logra cuando el ATmega detecta un evento (derrota, acierto en fila 4, acierto en fila 8) y ejecuta una función (sonido_perdedor, sonido_premio_chico, sonido_premio_grande) que

envía un pulso de duración específica por el pin PC2. El PIC recibe este pulso, lo mide, y reproduce la melodía vinculada.

Los tres niveles de dificultad (fácil, medio, difícil) se implementan en el ATmega, modificando dinámicamente el ancho (ancho) del bloque inicial y la velocidad (velocidad) de desplazamiento.

3.2. IMPLEMENTACIÓN

3.2.1. PROTEUS

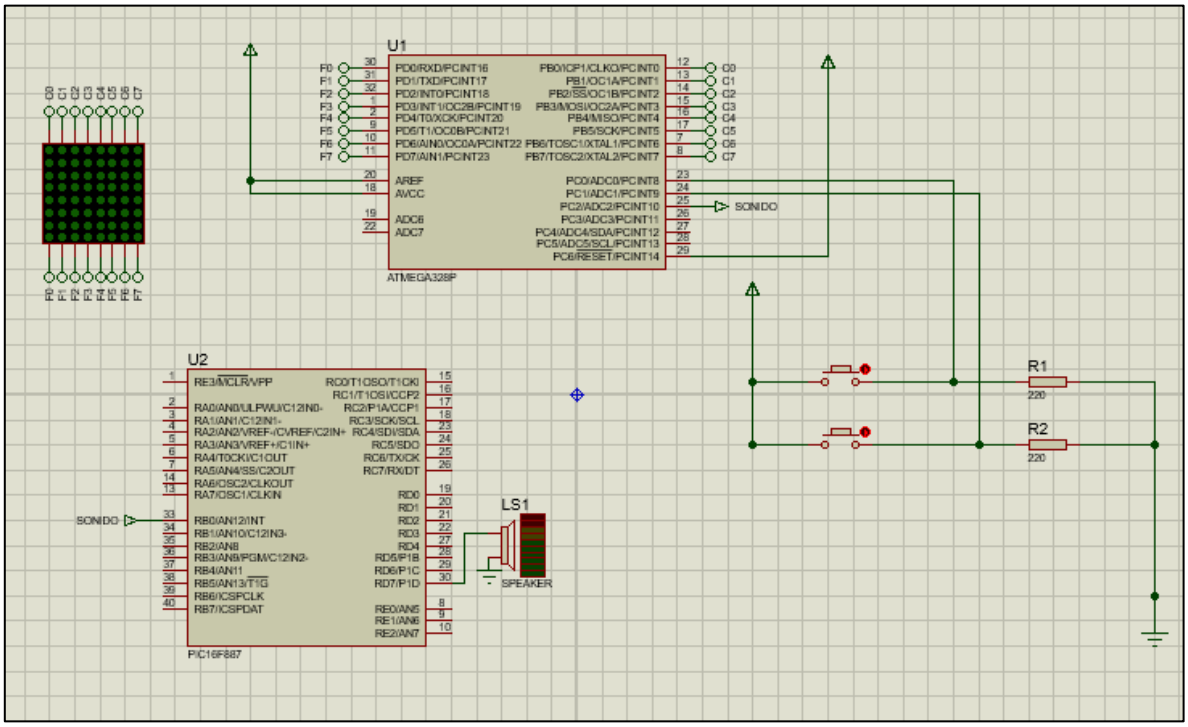

Figura 1. Diseño del circuito en Proteus

3.2.2. LENGUAJE DE DESRROLLO

En MikroC for PIC:

```
// MELODÍAS DE SONIDO

void Melody_Prize_Small() {
    Sound_Play(880, 90);   // A5
    Sound_Play(988, 90);   // B5
    Sound_Play(1174, 120); // D6
}
void Melody_Prize_Big() {
    Sound_Play(659, 150);  // E5
    Sound_Play(784, 150);  // G5
    Sound_Play(988, 200);  // B5
    Sound_Play(1318, 350); // E6 (brillante)
}


void Melody_Loser() {
    Sound_Play(330, 180);  // E4
    Sound_Play(294, 180);  // D4
    Sound_Play(247, 200);  // B3
    Sound_Play(196, 350);  // G3 (grave)
}
```

Figura 2. Configuración de las melodías de sonido

```
unsigned int medir_pulso_ms() {
    unsigned int tiempo = 0;

    // Esperar flanco de subida
    while (PORTB.F0 == 0);

    // Medir cuánto tiempo permanece en 1
    while (PORTB.F0 == 1) {
        Delay_ms(1);
        tiempo++;
        if (tiempo > 1000) break; // protección
    }

    return tiempo;
```

Figura 3. Configuración de la medición de pulsos (RB0)

```
void main() {

    // Configuración del hardware

    ANSEL = 0;
    ANSELH = 0;
    C1ON_bit = 0;
    C2ON_bit = 0;
    OSCCON = 0x71;     // 8 MHz interno

    TRISB.F0 = 1;      // RB0 como entrada (SONIDO0)
    TRISD.F7 = 0;      // RD7 salida de sonido
    PORTB.F0 = 0;

    Sound_Init(&PORTD, 7);    // Pin RD7 para el parlante
```

Figura 4. Función principal parte 1: configuración del hardware

```
    while (1) {

        unsigned int pulso = medir_pulso_ms();

        if (pulso <400) {
            // Entre 40-80 ms ? Perdedor
            Melody_Loser();
        }
        else if (pulso < 800) {
            // Entre 120-200 ms ? Premio chico
           Melody_Prize_Small();
        }
        else if (pulso < 1500){
            Melody_Prize_Big();
    }
}
```

Figura 5. Función principal parte 2: bucle principal

En Visual Studio Code + PlatformIO:

```
// ============================================================================
// VELOCIDAD CORRECTA (8MHz)
// ============================================================================
#define F_CPU 8000000UL

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

Figura 6. Configuración de la frecuencia del CPU e inclusión de librerías estándar de AVR.

```
// ============================================================================
// --- Definiciones de Pines ---
// ============================================================================
#define PIN_BOTON1 (1 << PC0)
#define PIN_BOTON2 (1 << PC1)
#define BOTON1_PRESIONADO (PINC & PIN_BOTON1)
#define BOTON2_PRESIONADO (PINC & PIN_BOTON2)

unsigned char FILAS[8] = {1,2,4,8,16,32,64,128};

unsigned char tablero_juego[8] = {0};
volatile int nivel_premio = 0;

// ----------- SONIDOS DIFERENCIADOS POR PULSOS -----------
void sonido_perdedor() { enviar_pulso(50); }        // C
void sonido_premio_chico() { enviar_pulso(150); }   // A
void sonido_premio_grande() { enviar_pulso(300); }  // B
void sonido_menu() { enviar_pulso(500); }  // Menu
```

Figura 7. Funciones del ATmega328P para enviar comandos de audio mediante pulsos de diferente duración.

```
unsigned char MENSAJE_STACKER[] = {
    0,0,78,82,82,114,0,0,
    0,2,2,126,126,2,2,0,
    0,0,126,18,18,126,0,0,
    0,0,124,66,66,66,0,0,
    0,0,126,16,16,40,70,0,
    0,0,126,82,82,82,0,0,
    0,0,126,18,50,110,0,0,
    0,0,0,0,0,0,0,0
};
int LARGO_MSG_STACKER = 64;

unsigned char CARA_FELIZ[8] = {
    0,  32, 78  ,64 ,64,    78, 32, 0

};

unsigned char CARA_TRISTE[8] = {
    0,  64, 46, 32  ,32 ,46,    64, 0

};

unsigned char NIVEL_FACIL[8]  = {0,64,64,126,126,68,64,0};
unsigned char NIVEL_NORMAL[8] = {0,92,92,84,116,116,0,0};
unsigned char NIVEL_DIFICIL[8]= {0,54,126,90,74,106,96,0};
```

Figura 8. Declaración de los patrones de bits para los símbolos y mensajes del juego.

```
void mostrar_estado_juego(unsigned char* tablero, int col_activa, unsigned char bloques_moviles) {
    for (int fila = 0; fila < 8; fila++) {
        PORTD = 0x00;

        unsigned char datos = 0;
        for (int col = 0; col < 8; col++) {
            int col_mapeada = 7 - col;
            unsigned char fuente = (col_mapeada == col_activa) ? bloques_moviles : tablero[col_mapeada
            if (fuente & (1 << fila)) datos |= (1 << col);
        }

        PORTB = ~datos;
```

Figura 9. Función de multiplexado que combina el tablero estático con los bloques en movimiento

```
void parpadear_pantalla(int veces, int retraso_ms) {
    for (int i = 0; i < veces; i++) {
        for (int f = 0; f < retraso_ms / 2; f++)
            for (int r = 0; r < 8; r++) {
                PORTD = FILAS[r];
                PORTB = 0x00;
                _delay_us(100);
            }

        for (int f = 0; f < retraso_ms / 2; f++)
            for (int r = 0; r < 8; r++) {
                PORTD = FILAS[r];
                PORTB = 0xFF;
                _delay_us(100);
            }
    }
    PORTD = 0;
}
```

Figura 10. Función parpadear_pantalla para alternar el estado de toda la matriz entre encendido/apagado

```
void desplazar_stacker(void) {
    while(1){
        for (int i = 0; i < LARGO_MSG_STACKER - 8; i++) {
            for (int rep = 0; rep < 40; rep++) {
                for (int r = 0; r < 8; r++) {
                    PORTD = 0;
                    PORTB = ~MENSAJE_STACKER[i + r];
                    PORTD = FILAS[r];
                    _delay_us(80);
                }
                if (BOTON1_PRESIONADO) return;
            }
        }
    }
}

void mostrar_cara(unsigned char *cara, int duracion_ms) {
    for (int t = 0; t < duracion_ms / 10; t++) {
        for (int fila = 0; fila < 8; fila++) {
            PORTD = 0;
            PORTB = ~cara[fila];
            PORTD = FILAS[7 - fila];
            _delay_us(300);
        }
    }
    PORTD = 0;
}
```

Figura 11. Funciones de visualización desplazar_stacker para la animación de título y mostrar_cara (ícono)

```
void parpadear_fila_premio(int fila, unsigned char patron_acierto) {
    for (int p = 0; p < 30; p++) {
        unsigned char patron = (p % 2 == 0) ? patron_acierto : 0xFF;

        for (int rep = 0; rep < 50; rep++) {
            for (int r = 0; r < 8; r++) {
                PORTD = 0;

                unsigned char datos = 0;
                for (int col = 0; col < 8; col++) {
                    int col_m = 7 - col;
                    unsigned char src = (col_m == fila) ? patron : tablero_juego[col_m];
                    if (src & (1 << r)) datos |= (1 << col);
                }

                PORTB = ~datos;
                PORTD = FILAS[7 - r];
                _delay_us(100);
            }

            if (BOTON2_PRESIONADO || BOTON1_PRESIONADO) return;
        }
    }
    PORTD = 0;
}
```

Figura 12. Función parpadear_fila_premio que resalta una fila específica, alternando su patrón para crear un parpadeo.

```
int seleccionar_nivel(void) {
    int nivel = 1; unsigned char* patron = NIVEL_FACIL;
    while(1) {
        for (int fila = 0; fila < 8; fila++) {
            PORTD = 0;
            PORTB = ~patron[fila];
            PORTD = FILAS[7 - fila];
            _delay_us(250);
        }
        PORTD = 0;

        if (BOTON1_PRESIONADO) {
            _delay_ms(20);
            while (BOTON1_PRESIONADO);

            nivel++;
            if (nivel > 3) nivel = 1;

            if (nivel == 1) patron = NIVEL_FACIL;
            else if (nivel == 2) patron = NIVEL_NORMAL;
            else patron = NIVEL_DIFICIL; }
        if (BOTON2_PRESIONADO) {
            _delay_ms(20);
            while(BOTON2_PRESIONADO);
            parpadear_pantalla(2, 50);
            return nivel;
        }
    }
}
```

Figura 13. Función seleccionar_nivel que gestiona el menú de selección de dificultad usando los pulsadores.

```
int ejecutar_juego(int nivel) {
    for (int i = 0; i < 8; i++) tablero_juego[i] = 0;
    unsigned char mascara = 0xFF;
    int velocidad = 25;

    unsigned char ancho;
    if (nivel == 1) ancho = 3;
    else if (nivel == 3) ancho = 1;
    else ancho = 3;

    for (int fila = 0; fila < 8; fila++) {
        int pos = 0, dir = 1;

        if (nivel == 2) {
            if (fila >= 3 && fila < 5) ancho = 2;
            if (fila >= 5) ancho = 1;
        }
```

Figura 14. Lógica principal del juego inicializando variables y ajustando el ancho del bloque según el nivel.

```
        while (1) {
            unsigned char patron_m = ((1 << ancho) - 1) << pos;

            for (int t = 0; t < velocidad; t++) {
                mostrar_estado_juego(tablero_juego, fila, patron_m);

                if (BOTON1_PRESIONADO) {
                    _delay_ms(10);
                    while (BOTON1_PRESIONADO)
                        mostrar_estado_juego(tablero_juego, fila, patron_m);
                    goto presionado;
                }
                _delay_ms(1);
            }

            pos += dir;
            if (pos + ancho > 8) { pos = 8 - ancho; dir = -1; }
            if (pos < 0) { pos = 0; dir = 1; }
            continue;

        presionado:;
            unsigned char acierto = patron_m & mascara;
            if (acierto == 0) {
                sonido_perdedor();
                mostrar_cara(CARA_TRISTE, 1500);
                return 0;
            }
```

Figura 15. Bucle de movimiento horizontal del bloque y lógica de validación de acierto (acierto) al presionar el botón.

```
            tablero_juego[fila] = acierto;
            mascara = acierto;

            ancho = 0;
            for (int b = 0; b < 8; b++)
                if (acierto & (1 << b)) ancho++;

            if (ancho == 0) {
                sonido_perdedor();
                mostrar_cara(CARA_TRISTE, 1500);
                return 0;
            }

            if (fila == 3) {
                parpadear_fila_premio(fila, acierto);
                if (BOTON2_PRESIONADO) {
                    sonido_premio_chico();
                    mostrar_cara(CARA_FELIZ, 1500);
                    return 4;
                }
            }
```

Figura 16. Lógica de acierto: actualización de la máscara, recálculo de ancho y gestión del "premio chico"

```
                if (fila == 7) {
                    sonido_premio_grande();
                    mostrar_cara(CARA_FELIZ, 2000);
                    return 8;
                }

                if (velocidad > 5) velocidad -= 4;
                break;
            }
        }

        sonido_premio_grande();
        mostrar_cara(CARA_FELIZ, 2000);
        return 8;
    }
```

Figura 17. Lógica de victoria para el "premio grande" e incremento de velocidad por cada fila superada.

```c
int main(void) {
    DDRB = 0xFF;
    DDRD = 0xFF;
    DDRC = 0x00;
    PORTC = 0x0F;

    sonido_init();
    desplazar_stacker();
    PORTD = 0;

    while (1) {
        if (BOTON1_PRESIONADO) {
            _delay_ms(20);
            while (BOTON1_PRESIONADO);

            int nivel = seleccionar_nivel();
            int res = ejecutar_juego(nivel);

            if (res == 4) mostrar_cara(CARA_FELIZ, 1500);
            if (res == 8) mostrar_cara(CARA_FELIZ, 2000);

            desplazar_stacker();
            PORTD = 0;
        }
    }
}
```

Figura 18. Función main que inicializa los puertos, muestra la animación de inicio y gestiona el bucle principal del juego.

4. MODELADO FINAL

El proyecto se implementó con éxito, logrando un prototipo funcional del videojuego arcade "Stacker" mediante la arquitectura de doble microcontrolador propuesta. La integración del ATmega328P para la lógica visual y el PIC16F887 como procesador de audio esclavo resultó ser una solución efectiva

La jugabilidad del prototipo es la siguiente: el usuario inicia el juego presionando el botón principal, lo que lo lleva a un menú para seleccionar uno de los tres niveles de dificultad. Una vez en el juego, el jugador debe presionar el botón principal para detener una fila de bloques que se mueve horizontalmente, intentando alinearla perfectamente con la fila estática inferior. Si falla por completo, el juego termina y se activa la melodía de "derrota". El jugador tiene la opción de reclamar un "premio chico" en la cuarta fila o continuar hasta la octava fila para ganar el "premio grande", donde cada resultado es acompañado por su correspondiente retroalimentación visual y auditiva. Se inserta un link redireccionando al GitHub donde se encuentra los archivos implementados al deber:

https://github.com/EstraRobert/Tarea2-Grupo2-SistemasEmbebidos/tree/main

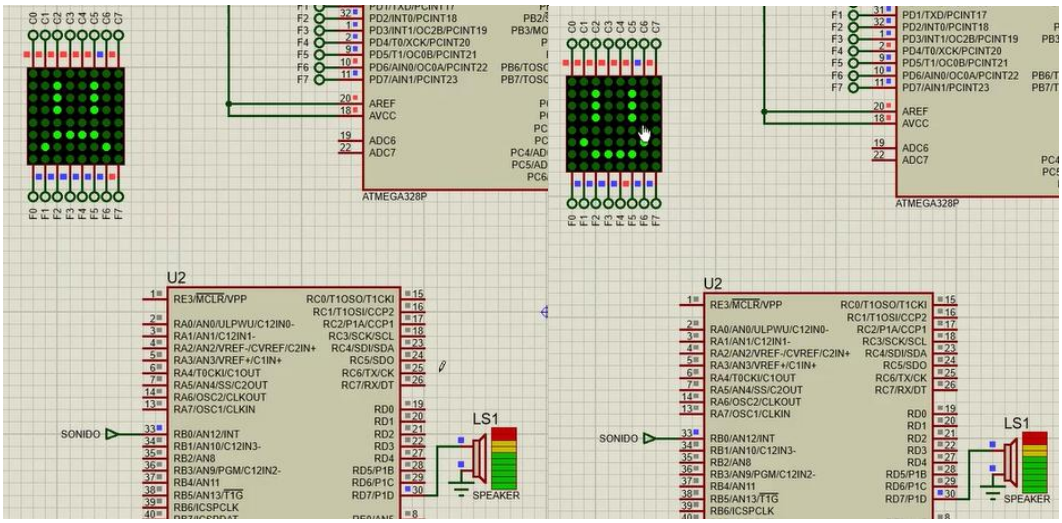

Figura 19. Forma de victoria/derrota en proteus.

5. CONCLUSIONES

Se implementó un prototipo funcional del juego "Stacker" mediante una arquitectura de doble microcontrolador (ATmega328P y PIC16F887).

Se validó un método de comunicación de un solo hilo, basado en ancho de pulso, como una solución eficaz para sincronizar eventos visuales y auditivos entre los dos procesadores.

Se comprobó la viabilidad y el funcionamiento correcto de la lógica de juego, incluyendo la multiplexación de la matriz LED y la gestión de niveles, mediante la simulación en Proteus.

6. RECOMENDACIONES

Implementar un protocolo de comunicación bidireccional más robusto, como I2C o UART, en lugar del sistema de pulsos. Esto permitiría al PIC16F887 enviar confirmaciones o datos de estado de vuelta al ATmega328P.

Ensamblar el circuito en un protoboard o PCB físico para evaluar el rendimiento del hardware en condiciones reales, especialmente la calibración del pin de audio (buzzer) y la respuesta visual de la matriz LED.