INFORME DE LABORATORIO 3:

Arduino vs Python vs C

Autores: *Santiago Giraldo Tabares, Ana María Velasco Montenegro*

Laboratorio de Electrónica Digital 3
Departamento de Ingeniería Electrónica y de Telecomunicaciones
Universidad de Antioquia

Resumen

En esta documento, se evaluarán diferentes flujos de programación y entornos de desarrollo para microcontroladores utilizando la Raspberry Pi Pico. Los flujos de programación realizados incluyen polling, interrupciones, y una combinación de ambos, implementados en los entornos Arduino, MicroPython, y SDK de C. Se realiza por medio del diseño de un Generador Digital de Señales (GDS) capaz de producir cuatro tipos de formas de onda, con parámetros de usuario para ajustar amplitud, nivel de DC y frecuencia a través de un teclado matricial. Además se medirá el rendimiento de cada enfoque y entorno de programación basándose en la frecuencia máxima de generación de señal estable, comparando también la facilidad de programación, el tamaño del programa y el uso de memoria RAM entre las diferentes implementaciones..

Palabras clave: *GPIO, Raspberry Pi Pico, Arduino, MicroPython, Generador Digital de Señales, Lenguaje C*

Introducción

Los sistemas embebidos son muy importantes en las aplicaciones tecnológicas debido a que ellos se encargan en realizar funciones específicas. Este laboratorio permite explorar las técnicas de programación fundamentales en sistemas embebidos mediante la implementación de un Generador Digital de Señales utilizando la Raspberry Pi Pico. Este dispositivo se programa para generar distintas formas de onda, manipuladas mediante una interfaz de usuario que permite cambiar la amplitud, el nivel de DC, y la frecuencia de la señal generada. Esta configuración permite entender las diferencias fundamentales entre los diferentes flujos de programación para posteriormente evaluar la eficacia de los entornos de desarrollo en la implementación de tareas específicas de hardware.

Marco teórico

Los sistemas embebidos desempeñan un papel muy importante debido a que son dispositivos que realizan tareas específicas de forma autónoma y eficiente. Estos sistemas, combinan hardware dedicado con software específicamente diseñado, se encuentran desde dispositivos móviles y electrodomésticos hasta automóviles y sistemas de transporte avanzados.

A continuación se desarrollarán tres aspectos impor-

tantes en el desarrollo de sistemas embebidos, los cuales son: los sistemas embebidos, los flujos de programación utilizados para controlarlos, y los entornos de desarrollo que facilitan la creación de soluciones eficientes. Cada uno de estos elementos es muy importante para la funcionalidad y eficiencia del dispositivo final, influyendo en su capacidad para responder a las demandas en tiempo real.

Se busca entender diferentes flujos de programación como el polling y las interrupciones, así como la selección del entorno de desarrollo y como estos pueden impactar el rendimiento de los sistemas embebidos.

0.0.1. Sistemas Embebidos:

Los sistemas embebidos son configuraciones de computación especializadas que se diseñan para realizar funciones específicas dentro de dispositivos o sistemas mayores. Estos sistemas integran hardware dedicado, como microprocesadores, microcontroladores o DSPs (Procesadores de Señal Digital), con software personalizado que está optimizado para controlar y monitorear sus operaciones de manera eficiente. La integración de estos sistemas en dispositivos más grandes permite optimizar recursos críticos como la potencia de procesamiento, el uso de memoria y el tiempo de respuesta. Además, los sistemas embebidos son cruciales para asegurar la funcionalidad continua y fiable, ya que a menudo operan en entornos donde el fallo o el retardo pueden tener consecuencias significativas.

0.0.2. Flujos de Programación en Sistemas Embebidos:

El flujo de programación determina cómo interactúa el software con el hardware para gestionar tareas y responder a eventos externos. Los principales métodos de programación en sistemas embebidos son: polling, interrupciones, y la combinación de ambos. Cada método tiene sus propias ventajas y desafíos, la diferencia entre ellos va ligada a la eficiencia y la eficacia de cada sistema sistema.

- **Polling:** El procesador ejecuta ciclos repetitivos en los que verifica el estado de uno o más dispositivos o condiciones. El polling puede causar una utilización ineficiente del procesador, ya que consume ciclos de procesamiento comprobando estados incluso cuando no ocurren cambios. Esto puede resultar en un uso elevado de energía y una reducción en la disponibilidad del procesador para otras tareas.
- **Interrupciones:** Las interrupciones permiten que los sistemas manejen eventos de manera más eficiente. En lugar de verificar continuamente el estado de un dispositivo, el procesador se configura para que sea notificado activamente (interrumpido) cuando un evento relevante ocurre. Esto libera al procesador para realizar otras tareas y mejora significativamente la eficiencia energética y de procesamiento. Sin embargo, el manejo de interrupciones puede complicar el diseño del software debido a la necesidad de procesar eventos de forma asíncrona y gestionar estados potencialmente no deterministas.
- **Polling e Interrupciones:** Algunos sistemas avanzados utilizan una combinación de polling e interrupciones para maximizar la eficiencia y la simplicidad. Este enfoque mixto permite utilizar interrupciones para eventos críticos que requieren atención inmediata, mientras que el polling se mantiene para condiciones menos críticas o cuando la simplicidad del código es una prioridad. Este método equilibra la carga del procesador y optimiza la respuesta del sistema ante diversos tipos de eventos.

0.0.3. Entorno de desarrollo:

La elección del entorno de desarrollo es muy importante para el éxito de cualquier proyecto, ya que esto influye directamente en la facilidad de programación, el tiempo de desarrollo, y la capacidad de optimización del hardware. Se mostrarán tres entornos de desarrollo populares utilizados en la programación de sistemas embebidos: Arduino, MicroPython, y el SDK de C.

- **Arduino:** Es ampliamente reconocido por su facilidad de uso y su comunidad activa, que proporciona muchas librerías y ejemplos. Arduino es ideal para principiantes y para el desarrollo rápido de prototipos debido a su entorno de programación amigable y hardware accesible. Sin embargo, puede ser limitante para aplicaciones avanzadas que requieren un control riguroso del hardware o una optimización profunda.
- **MicroPython:** Este entorno eleva el nivel de abstracción de la programación de microcontroladores al permitir el uso del lenguaje Python, que es conocido por su legibilidad y facilidad de uso. MicroPython es particularmente accesible para desarrolladores que ya están familiarizados con Python, aunque puede ser menos eficiente en

términos de rendimiento comparado con lenguajes más cercanos al hardware como C.

- **SDK de C:** Proporciona el máximo control sobre el hardware y permite una optimización exhaustiva, lo cual es crítico en aplicaciones industriales y de sistemas embebidos donde la eficiencia y la precisión son fundamentales. Aunque requiere una comprensión más profunda de la arquitectura del microcontrolador y de la programación de bajo nivel, el SDK de C es indispensable para desarrollar aplicaciones que demandan el máximo rendimiento del hardware.

Procedimiento:

Para este laboratorio, la estrategia empieza con un análisis del problema a solucionar, que consiste en desarrollar un Generador Digital de Señales (GDS) capaz de producir diversas formas de onda controladas a través de una interfaz de usuario. El dispositivo permite al usuario seleccionar y modificar las características de la señal generada, como la amplitud, el nivel de DC, y la frecuencia, utilizando un teclado matricial. Además, un botón permite cambiar secuencialmente entre diferentes tipos de onda: senoidal, triangular, diente de sierra y cuadrada.

0.0.1. Configuración del hardware y entorno de desarrollo:

Involucra la preparación y configuración de la Raspberry Pi Pico junto con todos los componentes necesarios como el teclado matricial y los dispositivos de salida para visualización y verificación de las señales generadas.

0.0.2. Desarrollo del Firmware:

Se dividió en cinco flujos principales de programación que corresponden a diferentes combinaciones de técnicas de polling e interrupciones. Cada flujo fue implementado en diferentes entornos de desarrollo: Arduino, MicroPython y el SDK de C.

- **Arduino** Se usa el DAC integrado o conectado para convertir valores digitales a señales analógicas, controlando así la amplitud, el nivel de DC y la frecuencia de las formas de onda generadas. Arduino es ideal para principiantes y proyectos donde se requiere una actualización continua de los estados de entrada.

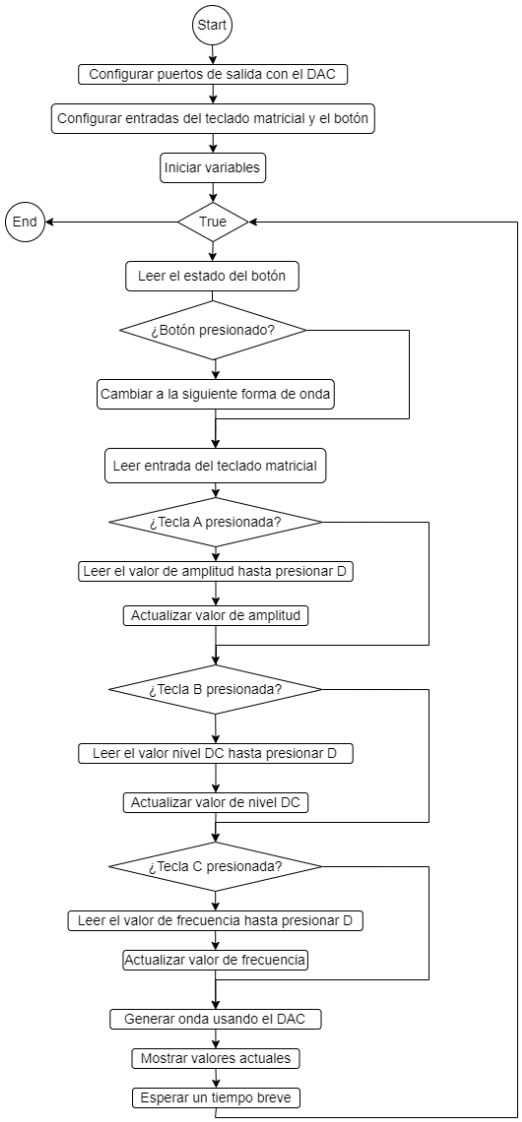

Figura **0-1**: Diagrama de flujo Arduino

- **MicroPython** Se usa la accesibilidad del Python, se maneja el DAC para traducir las instrucciones digitales en acciones analógicas que alteran las características de las formas de onda. Esta implementación se destaca por su simplicidad y eficacia, permitiendo a los usuarios con conocimientos básicos de Python manipular hardware de forma intuitiva y efectiva.

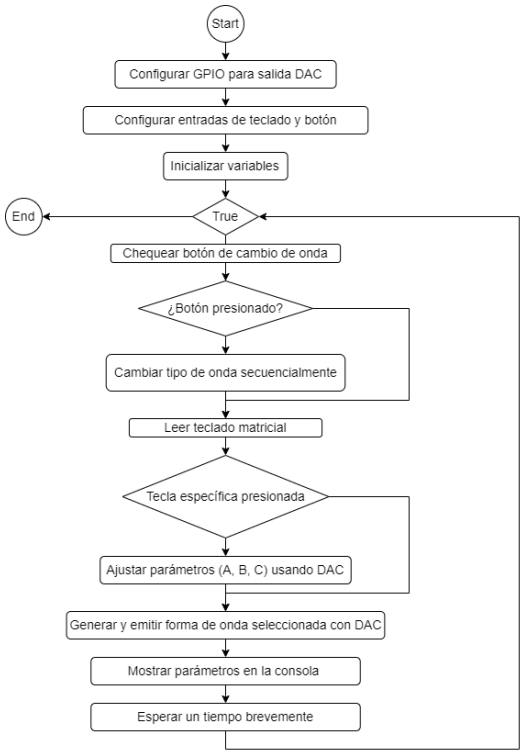

Figura **0-2**: Diagrama de flujo Micropython

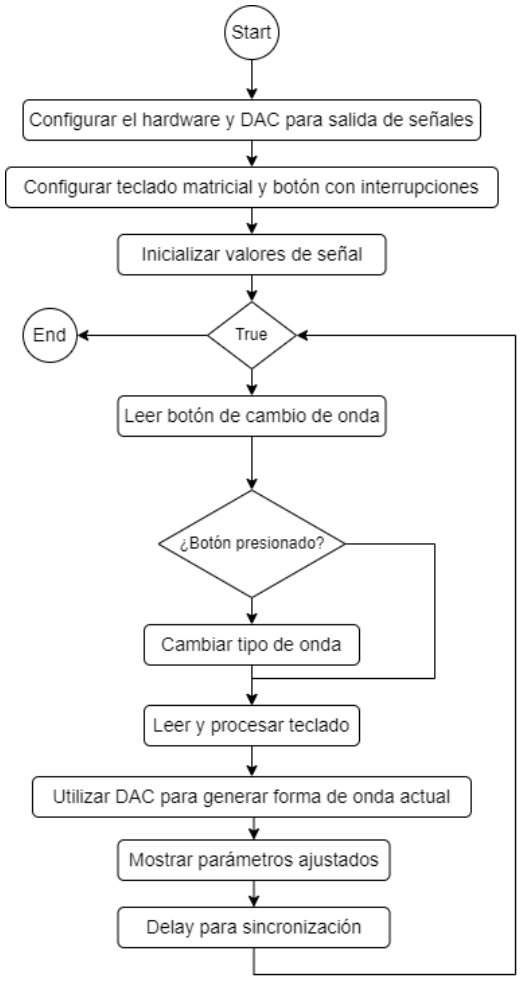

Figura **0-3**: Diagrama de flujo C con Polling

- **C con Polling** Se emplea el DAC para generar precisamente las señales requeridas basadas en los parámetros definidos por el usuario. Este método es adecuado para sistemas embebidos donde se requiere un control preciso del hardware, maximizando la eficiencia y precisión de las salidas analógicas.

- **C con Interrupciones** El DAC juega un papel crucial al convertir instantáneamente las instrucciones digitales en señales analógicas cada vez que se recibe una interrupción, lo que permite una respuesta rápida y eficiente a las interacciones del usuario. Este método es ideal para aplicaciones que requieren una respuesta rápida y mínima latencia en la respuesta a eventos.

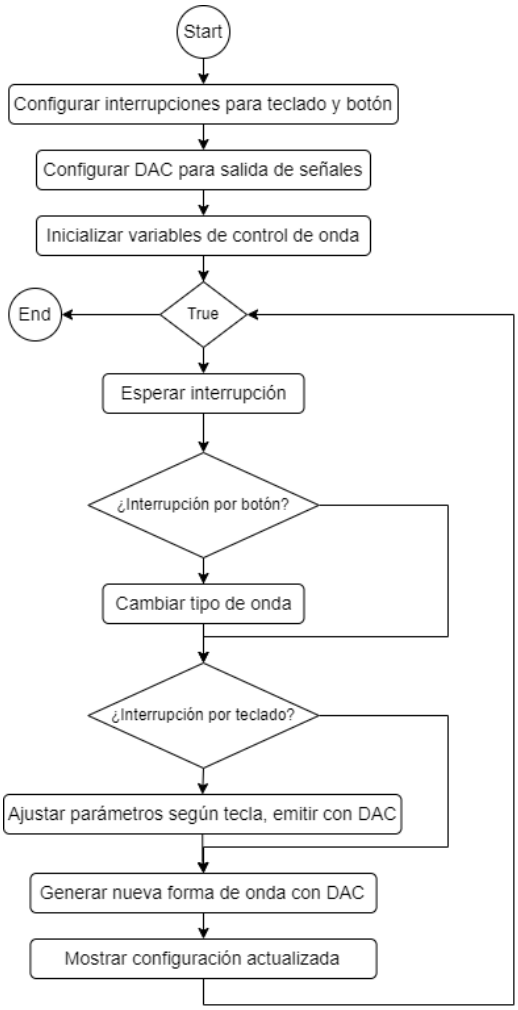

Figura **0-4**: Diagrama de flujo C con Interrupciones

- **C con Polling + Interrupciones** Al integrar el DAC, se asegura que tanto las entradas de alta prioridad (gestionadas por interrupciones) como las de menor prioridad (manejadas a través de polling) se procesen eficazmente, convirtiendo las señales digitales en analógicas de manera precisa. Este enfoque híbrido es excelente para sistemas que necesitan balancear entre la eficiencia del procesamiento y la complejidad del manejo de eventos.

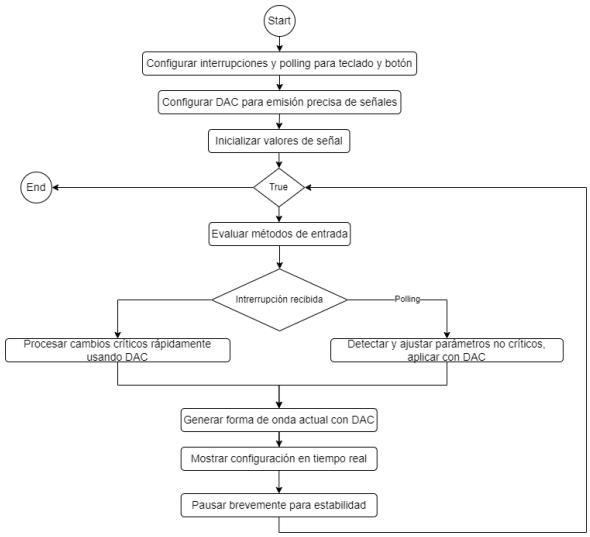

Figura **0-5**: Diagrama de flujo C con Polling + Interrupciones

0.0.3. Módulos a usar:

- **Módulo de configuración de hardware:** Configura el hardware necesario, incluyendo GPIOs para el teclado matricial, el botón de cambio de onda y el DAC.

  Funcionalidad: La configuración de pines como entradas o salidas y la inicialización del DAC para la generación de señales
- **Módulo de gestión de entrada:** Este módulo gestiona las entradas del teclado matricial y del botón de cambio de onda. **Funcionalidad:** La lectura de estados del teclado y el botón, el debouncing de entradas para evitar lecturas erróneas y la conversión de entradas en comandos para ajustar parámetros o cambiar formas de onda.
- **Módulo de generación de formas de onda:** Genera las diferentes formas de onda basadas en los parámetros actuales usando el DAC.

  Funcionalidad: La generación de onda senoidal, triangular, diente de sierra y cuadrada y ajuste de amplitud, frecuencia y nivel DC basado en las entradas del usuario.
- **Módulo de visualización:** Este módulo muestra información relevante al usuario a través de una interfaz serial. **Funcionalidad:** Mostrar el tipo de onda actual, ampli-

tud, nivel DC y frecuencia y la actualización en tiempo real de los parámetros ajustados.

0.0.4. Circuito implementado:

Se puede observar el circuito completo implementado con todos los elementos como resistencias, capacitor (104), pulsador, DAC, teclado matricial, amplificación operacional y la Raspberry Pi Pico.

Se usa el DAC0808 para convertir señales digitales en analógicas, las resistencias para establecer puntos de referencia y ganancias específicas, el amplificador operacional para ajustar y filtrar/amplificar la señal analógica, y el condensador para estabilizar el voltaje y filtrar ruido.

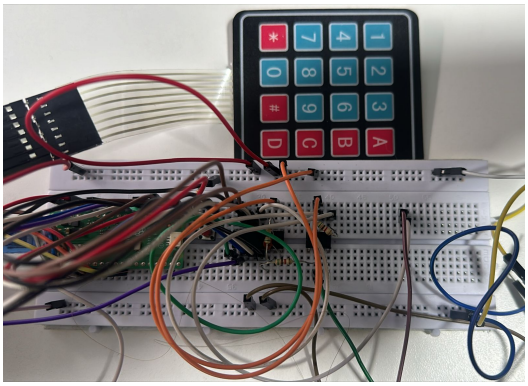

Figura **0-6**: Circuito con DAC y teclado matricial

Se tienen estos dos esquemáticos, el primero es la implementación del DAC y el segundo, el circuito implementado para la práctica

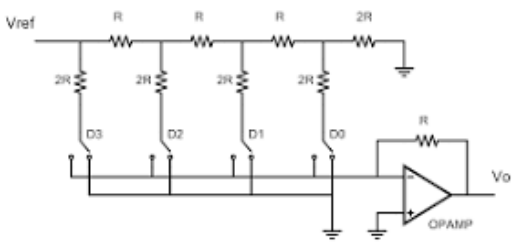

Figura **0-7**: DAC con resisencias

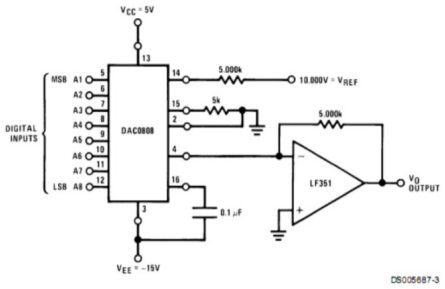

Figura **0-8**: Esquemático

0.0.5. Código de Arduino

Este código de Arduino está diseñado para controlar un Generador Digital de Señales (GDS) que produce varias formas de onda analógicas por medio de un DAC. Se incluye una librería matemática para utilizar funciones esenciales en el cálculo de las formas de onda. Los parámetros predeterminados y los rangos permitidos para la amplitud y la frecuencia se establecen a través de constantes. También se define un valor máximo de salida para el DAC de 8 bits y un voltaje de referencia que el DAC utilizará para las conversiones digital-analógico.

El arreglo bidimensional keys y los pines rowPins y colPins están configurados para manejar la entrada del teclado matricial de 4x4, permitiendo cambiar los parámetros de la señal. El botón de cambio de onda, asociado a un pin específico, permite cambiar entre diferentes formas de onda. Los pines conectados al DAC se definen en un array, preparándolos para la salida de la señal.

Una enumeración Waveform facilita el cambio entre diferentes tipos de ondas. Las variables globales almacenan los valores actuales de amplitud, frecuencia y desplazamiento de DC, los cuales son muy importantes para la generación de señales.

La función setup_gpio() inicializa los pines del DAC como salidas y el botón como entrada con resistencia de pull-up. La función write_dac() se encarga de convertir un valor digital en la activación correspondiente de los pines del DAC, generando así la salida analógica deseada.

El código en el método setup() inicia la comunicación serial y muestra los valores iniciales de los parámetros de la señal. La lectura de las entradas

del teclado y la actualización de los parámetros de la señal se manejan a través de las funciones readAmplitude(), readFrequency(), y readDCOffset(), las cuales permiten introducir nuevos valores que se validan y aplican a la señal generada.

La función changeWaveform() ajusta el tipo de forma de onda en respuesta a las presiones del botón y anuncia el cambio por el serial. En el bucle principal loop(), se leen las entradas del teclado y se verifica si el botón ha sido presionado, ajustando los parámetros y generando la forma de onda correspondiente. Una serie de operaciones matemáticas son utilizadas para calcular el valor correcto de la señal en cada punto del tiempo, basado en la forma de onda actual.

El código permite visualizar las características de la señal y ofrece una visualización en tiempo real de cualquier cambio a través de la consola serial. El código se adjuntará en los anexos.

```
const byte ROWS = 4;            // Cuatro filas
const byte COLS = 4;            // Cuatro columnas
char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
byte rowPins[ROWS] = {15, 14, 13, 12}; // Pines de fila en Raspberry Pi Pico
byte colPins[COLS] = {11, 10, 9, 8};   // Pines de columna en Raspberry Pi Pico

const byte waveformButtonPin = 16;     // Pin del botón

byte dacPins[8] = {0, 1, 2, 3, 4, 5, 6, 7}; // Pines DAC en Raspberry Pi Pico

enum Waveform { SINE, SQUARE, SAWTOOTH, TRIANGULAR };
Waveform currentWaveform = SINE;

float amplitude = AMPLITUDE_DEFAULT; // Amplitud predeterminada en milivoltios
float frequency = 400;     // Frecuencia predeterminada en Hz
float dcOffset = (AMPLITUDE_DEFAULT) / 2.0; // Desplazamiento DC predeterminado
```

Figura **0-9**: Código Arduino

```
void setup_gpio() {
    for (int i = 0; i < 8; i++) {
        pinMode(dacPins[i], OUTPUT);
    }
    pinMode(waveformButtonPin, INPUT_PULLUP);
}

void write_dac(uint8_t value) {
    for (int i = 0; i < 8; i++) {
        digitalWrite(dacPins[i], (value >> i) & 1);
    }
}

void changeWaveform() {
    currentWaveform = static_cast<Waveform>((currentWaveform + 1) % 4);
    switch (currentWaveform) {
        case SINE:
            Serial.println("Forma de onda cambiada a: Seno");
            break;
        case SQUARE:
            Serial.println("Forma de onda cambiada a: Cuadrada");
            break;
        case SAWTOOTH:
            Serial.println("Forma de onda cambiada a: Diente de sierra");
            break;
```

Figura **0-10**: Código Arduino

```
char getKeypadInput() {
    char key = '\0';

    // Establecer todos los pines de columna como INPUT_PULLUP
    for (byte col = 0; col < COLS; col++) {
        pinMode(colPins[col], INPUT_PULLUP);
    }

    // Escanear los pines de fila en busca de una entrada LOW
    for (byte row = 0; row < ROWS; row++) {
        pinMode(rowPins[row], OUTPUT);
        digitalWrite(rowPins[row], LOW);

        for (byte col = 0; col < COLS; col++) {
            if (digitalRead(colPins[col]) == LOW) {
                // Tecla detectada, determinar la tecla correspondiente
                key = keys[row][col];
                delay(50); // Retraso de rebote
                while (digitalRead(colPins[col]) == LOW) {} // Esperar la liberación de la tecla
            }
        }

        // Restablecer la fila actual
        digitalWrite(rowPins[row], HIGH);
        pinMode(rowPins[row], INPUT);
    }
```

Figura **0-11**: Código Arduino

0.0.6. Código de Micropython

El código es en MicroPython para un Generador Digital de Señales (GDS) operado por la Raspberry Pi Pico. El GDS es capaz de generar cuatro formas de onda diferentes: seno, triangular, diente de sierra y cuadrada. Se puede cambiar la forma de onda presionando un botón y ajustar la amplitud, el nivel de DC y la frecuencia utilizando un teclado matricial.

Primero, se configuran los pines de la Raspberry Pi Pico para interactuar con el teclado matricial. Las columnas del teclado se configuran como salidas y las filas como entradas con resistencias de pull-up.

El script define una matriz key_map que mapea cada tecla a su posición correspondiente en el teclado matricial. Para la señalización, se utiliza un canal PWM conectado al pin 15, y se establece una frecuencia inicial de 10 Hz.

Los valores predeterminados de amplitud, nivel de DC y frecuencia se establecen en 1000 mV, 500 mV y 10 Hz, respectivamente. Además, se declara una lista wave_forms que contiene las formas de onda disponibles y un índice current_waveform_index para llevar un registro de la forma de onda actual.

El botón para cambiar las formas de onda se monitorea mediante una interrupción configurada en el pin 16. La función button_press_handler maneja esta interrupción, rotando secuencialmente a través de las formas de onda cada vez que se detecta un flanco ascendente.

La función read_keypad lee las entradas del teclado matricial y devuelve la tecla que ha sido presionada. get_numeric_input utiliza esta función

para obtener entradas numéricas del usuario, asegurándose de que los valores ingresados estén dentro de los límites definidos.

La función setup_parameters invoca get_numeric_input para establecer la amplitud, el nivel de DC y la frecuencia basados en las entradas del teclado.

La función generate_wave genera la forma de onda seleccionada calculando el ciclo de trabajo (duty cycle) para el PWM basado en la forma de onda actual y los parámetros establecidos. Para evitar la división por cero, se usa un valor seguro para la frecuencia (safe_frequency). El ciclo de trabajo se ajusta para valores entre 0 y 65535, que es el rango aceptado por pwm.duty_u16.

Finalmente, la función main establece los parámetros iniciales y luego entra en un bucle continuo, generando la forma de onda y actualizando los parámetros si es necesario. Los parámetros actuales de la señal se imprimen cada segundo, proporcionando una retroalimentación constante al usuario.

```python
def button_press_handler(pin):
    global current_waveform_index, wave_forms
    current_waveform_index = (current_waveform_index + 1) % len(wave_forms)
    print("Cambio de forma de onda a:", wave_forms[current_waveform_index])

button.irq(trigger=Pin.IRQ_RISING, handler=button_press_handler)

def read_keypad():
    for col_num, col in enumerate(cols):
        col.value(0)
        for row_num, row in enumerate(rows):
            if not row.value():
                utime.sleep_ms(20)
                if not row.value():
                    col.value(1)
                    return key_map[row_num][col_num]
        col.value(1)
    return None
```

Figura **0-12**: Código Micropython

```python
def get_numeric_input(prompt, min_val, max_val):
    print(prompt)
    value = ''
    while True:
        key = read_keypad()
        if key and key.isdigit():
            value += key
            print(value)  # Echo the digit
        elif key == 'D' and value:  # Complete on 'D' press
            numeric_value = int(value)
            if numeric_value < min_val or numeric_value > max_val:
                print("Valor fuera de rango. Intente nuevamente.")
                value = ''  # Reset value if out of range
            else:
                return numeric_value
        utime.sleep(0.1)
```

Figura **0-13**: Código Micropython

```python
def generate_wave():
    global pwm, amplitude, dc_offset, frequency, current_waveform_index, wave_forms
    # Asegurarse de que frequency nunca sea cero para evitar división por cero
    safe_frequency = max(frequency, 1)
    step = 0
    while step < 360:
        # Selecciona la forma de onda basada en current_waveform_index
        wave_form = wave_forms[current_waveform_index]

        # Calcula el valor de duty para la forma de onda actual
        if wave_form == 'sine':
            duty = amplitude * (math.sin(math.radians(step)) / 2 + 0.5) + dc_offset
        elif wave_form == 'triangle':
            duty = (2 * amplitude / math.pi) * math.asin(math.sin(math.radians(step))) + dc_offset
        elif wave_form == 'sawtooth':
            duty = ((-2 * amplitude / math.pi) * math.atan(1/math.tan(math.radians(step / 2)))) + dc_offset
        elif wave_form == 'square':
            duty = amplitude if step < 180 else 0
            duty += dc_offset

        # Escala el valor de duty para que esté entre 0 y 65535
        pwm.duty_u16(int((duty / 3300) * 65535))
```

Figura **0-14**: Código Micropython

0.0.7. Código de C con Polling

Este código permite mostrar formas de onda como seno, cuadrada, diente de sierra y triangular mediante interacciones con un teclado matricial de 4x4 y un botón de cambio de forma de onda. El código inicializa GPIOs para las entradas del teclado y el botón, y configura un DAC de 8 bits para la salida de señales analógicas. Utiliza funciones específicas para leer las entradas del teclado matricial, permitiendo al usuario ajustar la amplitud, frecuencia y desplazamiento de DC. Una función de interrupción maneja el cambio de forma de onda cada vez que se presiona el botón, ciclando entre las formas de onda disponibles. Otra función esencial del código es write_dac, que convierte valores digitales a analógicos enviando datos a través de los GPIOs conectados al DAC. Adicionalmente, se utilizan librerías estándar como math.h para cálculos matemáticos necesarios para generar las ecuaciones de las formas de onda, y string.h y stdlib.h para manipular las entradas del teclado y convertirlas en valores numéricos. El bucle principal del programa se encarga de invocar estas funciones continuamente para ajustar los parámetros de las señales y generar las ondas según los parámetros configurados, imprimiendo actualizaciones y gestionando la interacción del usuario de manera eficiente. Esta estructura permite una correcta manipulación de hardware y software.

```c
char keys[ROWS][COLS] = {{'1','2','3','A'},{'4','5','6','B'},{'7','8','9','C'},{'*','0','#','D'}}; ///< 4x4 array, containing each of the

uint rowPins[ROWS] = {18, 19, 20, 21}; ///< Rows pinout (GPIOs) in the RP2040
uint colPins[COLS] = {22, 26, 27, 28};  ///< Columns pinout (GPIOs) in the RP2040

const uint waveformButtonPin = 16;   ///< Pushbutton GPIO.

uint dacPins[8] = {0, 1, 2, 3, 4, 5, 6, 7}; ///< output GPIOs to be connected to the DAC. Ordered from LSB to MSB.

typedef enum {
    SINE, ///< Sinusoidal wave. Sin(2*pi*ft). centered aroung its DC offset
    SQUARE, ///< Square wave, also called pulsed wave, symetric (50% duty cycle) centered aroung its DC offset
    SAWTOOTH, ///< Sawtooth wave. Rising time matches the period, while falling time goes to zero. centered aroung its DC offset
    TRIANGULAR ///< Triangular wave. Symetric (rising time equals 50% of period, falling time equals 50% of period). Centered aroung its D
} Waveform; ///< Predefined signal waveforms.

Waveform currentWaveform = SINE; ///< Current waveform produced by the signal. In this line, set the default Signal waveform as Sinewave.

float amplitude = AMPLITUDE_DEFAULT; ///< Set current amplitude to DEFAULT value
float frequency = 10;   ///< Set current frequency to  DEFAULT value
float dcOffset = (AMPLITUDE_MIN + AMPLITUDE_MAX) / 2.0; ///< Set current DC offset to DEFAULT value.
```

Figura **0-15**: Código C con Polling

Figura **0-16**: Código C con Polling

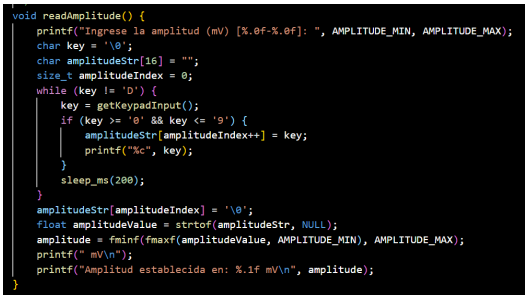

Figura **0-17**: Código C con Polling

0.0.8. Código de C con Interrupciones

Este código utiliza interrupciones para cambiar entre diversas formas de onda y para registrar entradas de un teclado matricial, este programa permite ajustar amplitud, frecuencia y desplazamiento DC. El circuito incluye un botón en GP16 para alternar formas de onda y conexiones de teclado en varios pines GPIO. Las bibliotecas utilizadas son pico/stdlib.h, hardware/gpio.h, y hardware/irq.h. El programa mejora la eficiencia mediante el manejo de interrupciones para todas las entradas de usuario y está diseñado para ser reactivo y eficiente en tiempo real. Desarrollado por Santiago Giraldo Tabares y Ana María Velasco Montenegro en abril de 2024, el código se ofrece sin licencia para uso libre, destacando su accesibilidad y adaptabilidad para diversos usos educativos o de desarrollo.

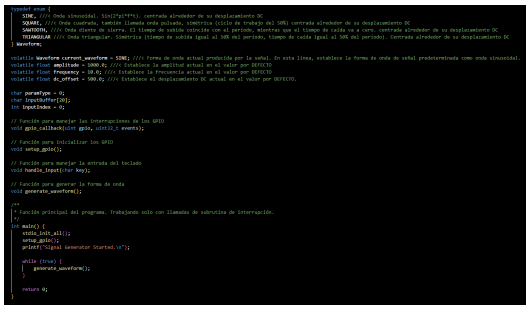

Figura **0-18**: C con Interrupciones

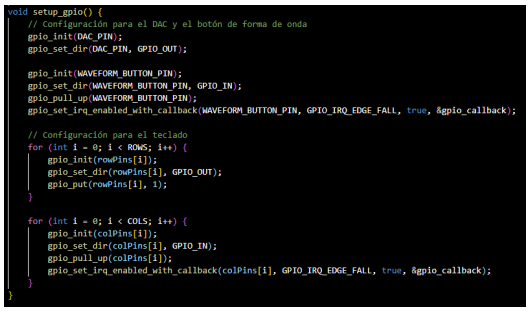

Figura **0-19**: C con Interrupciones

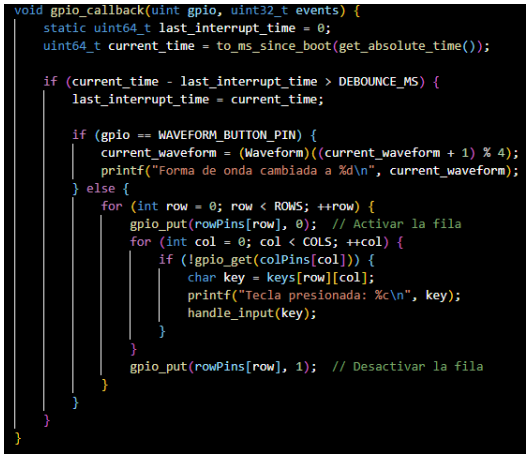

Figura **0-20**: C con Interrupciones

0.0.9. Código de C con Polling + Interrupciones

Este código integra métodos de interrupción y polling para gestionar entradas y producir señales analógicas. Utiliza interrupciones para manejar las entradas del botón que cambia las formas de onda,

asignado al GPIO16. Esta configuración de interrupción permite que el sistema responda inmediatamente cuando el usuario presiona el botón para alternar entre las formas de onda seno, cuadrada, diente de sierra y triangular. Este método de interrupción mejora la eficiencia del sistema al eliminar la necesidad de verificar constantemente el estado del botón en un ciclo de polling, liberando recursos del procesador para otras tareas.

```c
void waveformButtonCallback(uint gpio, uint32_t events) {
    // Registrar el tiempo actual
    uint64_t currentTime = time_us_64();

    // Verificar si ha pasado el tiempo de debounce
    if (currentTime - lastButtonPressTime >= DEBOUNCE_DELAY_US) {
        // Cambiar la forma de onda cuando se detecte una interrupción en el botón
        if (gpio == waveformButtonPin && events == GPIO_IRQ_EDGE_RISE) {
            changeWaveform();
        }

        // Actualizar el tiempo de la última pulsación del botón
        lastButtonPressTime = currentTime;
    }
}
```

Figura **0-22**: C con Polling + Interrupciones

El polling se usa para manejar las entradas del teclado matricial. El programa configura las filas y columnas del teclado como salidas y entradas respectivamente y realiza sondeos activos para detectar qué teclas son presionadas. Este método es necesario para capturar los valores de amplitud, frecuencia y desplazamiento de DC que el usuario desea ajustar. Las funciones readAmplitude(), readFrequency(), y readDCOffset() dependen de esta técnica de sondeo para obtener los datos introducidos por el usuario a través del teclado, procesarlos y aplicar los cambios a la señal generada. Aunque el sondeo es menos eficiente desde el punto de vista del uso de la CPU en comparación con las interrupciones, es esencial para el manejo de múltiples entradas que requieren una verificación constante del estado del teclado matricial. Este enfoque dual asegura que el generador de señales pueda responder rápidamente a cambios críticos a través de interrupciones, mientras gestiona entradas más complejas y configuraciones a través del polling.

```c
char getKeypadInput() {
    char key = '\0';
    for (uint col = 0; col < COLS; col++) {
        gpio_init(colPins[col]);
        gpio_set_dir(colPins[col], GPIO_IN);
        gpio_pull_up(colPins[col]);
    }

    for (uint row = 0; row < ROWS; row++) {
        gpio_init(rowPins[row]);
        gpio_set_dir(rowPins[row], GPIO_OUT);
        gpio_put(rowPins[row], 0);

        for (uint col = 0; col < COLS; col++) {
            if (!gpio_get(colPins[col])) {
                key = keys[row][col];
                sleep_ms(50); // Debounce delay
                while (!gpio_get(colPins[col])) {} // Wait for key release
            }
        }

        gpio_put(rowPins[row], 1);
        gpio_set_dir(rowPins[row], GPIO_IN);
    }

    return key;
}
```

Figura **0-23**: C con Polling + Interrupciones

Ondas en el osciloscopio

```c
char keys[ROWS][COLS] = {{'1','2','3','A'},{'4','5','6','B'},{'7','8','9','C'},{'*','0','#','D'}}; ///< 4x4 array, containing each of the

uint rowPins[ROWS] = {18, 19, 20, 21}; ///< Rows pinout (GPIOs) in the RP2040
uint colPins[COLS] = {22, 26, 27, 28};   ///< Columns pinout (GPIOs) in the RP2040

const uint waveformButtonPin = 16;     ///< Pushbutton GPIO.

uint dacPins[8] = {0, 1, 2, 3, 4, 5, 6, 7}; ///< output GPIOs to be connected to the DAC. Ordered from LSB to MSB.

typedef enum {
    SINE, ///< Sinusoidal wave. Sin(2*pi*f*t). centered aroung its DC offset
    SQUARE, ///< Square wave, also called pulsed wave, symetric (50% duty cycle) centered aroung its DC offset
    SAWTOOTH, ///< Sawtooth wave. Rising time matches the period, while falling time goes to zero. centered aroung its DC offset
    TRIANGULAR ///< Triangular wave. Symetric (rising time equals 50% of period, falling time equals 50% of period). Centered aroung its D
} Waveform; ///< Predefined signal waveforms.

Waveform currentWaveform = SINE; ///< Current waveform produced by the signal. In this line, set the default Signal waveform as Sinewave.

float amplitude = AMPLITUDE_DEFAULT; ///< Set current amplitude to DEFAULT value
float frequency = 10; ///< Set current frequency to  DEFAULT value
float dcOffset = (AMPLITUDE_MIN + AMPLITUDE_MAX) / 2.0; ///< Set current DC offset to DEFAULT value.

bool buttonState = false; ///< Set current state of the push-button as NOT-PRESSED.
uint64_t lastButtonPressTime = 0; ///< With debouncing purposes. Record the time when a button was pressed.
```

Figura **0-21**: C con Polling + Interrupciones

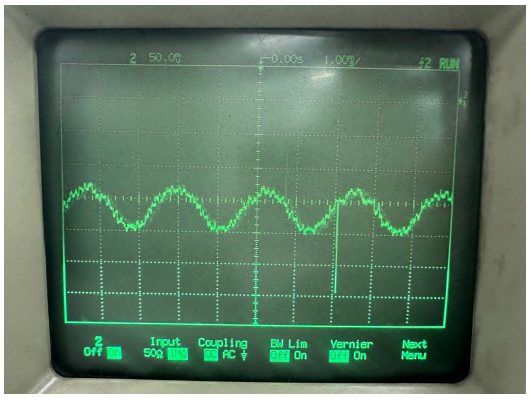

Figura **0-24**: Onda seno

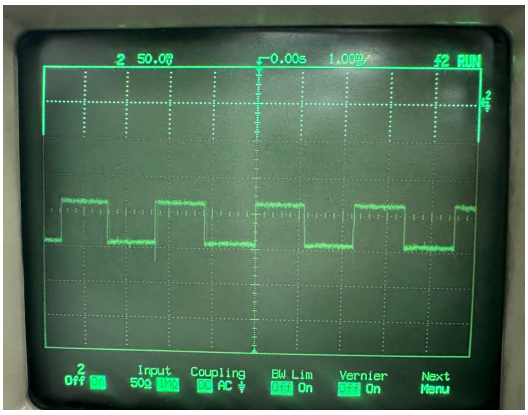

Figura **0-25**: Onda cuadrada

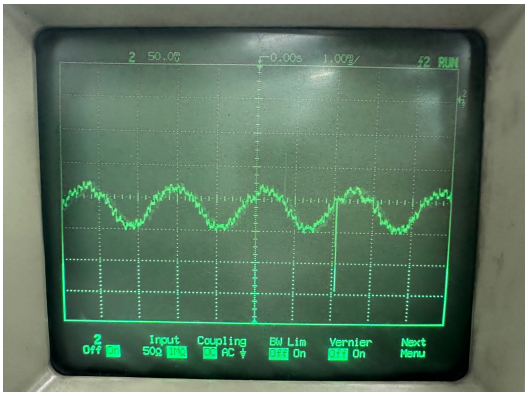

Figura **0-26**: Onda triangular

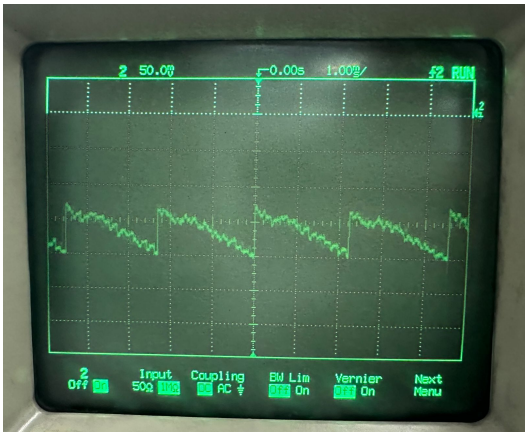

Figura **0-27**: Onda diente de sierra

Obstáculos en la implementación

Durante la implementación de este algoritmo surgieron una serie de dificultades asociadas

1. **Arduino con polling:** Arduino es fácil de usar pero puede no ser el más eficiente en términos de manejo de recursos como CPU y memoria, lo cual es crítico en sistemas embebidos. **Limitaciones de hardware:** Arduino puede ser limitado en términos de velocidad y capacidades de procesamiento en comparación con otras plataformas más robustas.
2. **MicroPython con polling:** MicroPython tiende a ser menos eficiente que C en términos de velocidad y uso de recursos, lo que puede ser un problema al manejar altas frecuencias o múltiples tareas simultáneamente.
3. **Dependencia de la interpretación de Python:** La naturaleza interpretada de Python puede introducir latencias adicionales y menor previsibilidad en la ejecución del tiempo real.
4. **C con polling:** Programar en C requiere un manejo manual de muchos aspectos de la ejecución, incluyendo la memoria y el manejo de dispositivos, lo cual puede introducir errores difíciles de detectar. C es un lenguaje de bajo nivel que puede ser desafiante para los programadores sin experiencia en sistemas embebidos.
5. **C con interrupciones:** Configurar correctamente las interrupciones en C puede ser complejo y propenso a errores, especialmente en sistemas con múltiples fuentes de interrupción. También el uso extensivo de interrupciones puede llevar a problemas de sincronización y condiciones de carrera si no se manejan cuidadosamente.
6. **C usando polling + interrupciones:** Combinar polling e interrupciones requiere un diseño cuidadoso para evitar conflictos y asegurar que el sistema responda de manera eficiente sin perder eventos. Es difícil de depurar debido a la naturaleza asincrónica de las interrupciones combinadas con el polling regular.

7. **Problemas de hardware:** Es complicado garantizar y controlar el hardware ya que muchas veces se ve afectado por agentes externos.

Discusión de resultados

0.0.1. Calificación

Al comparar la facilidad de programación entre los 5 flujos de diseño, se le asigna una calificación de 1 a 10 al nivel de facilidad de cada flujo de diseño, siendo 1 muy difícil y 10 muy fácil.

1. **Arduino con polling: Calificación: 8** Arduino tiene una amplia comunidad, documentación extensa, y un entorno de desarrollo simple. El manejo de polling es directo pero puede no ser el más eficiente para tareas complejas. Se reduce la dificultad respecto al de C, debido a que no toca hacer CMake, la sintaxis de Arduino es más amigable y adicionalmente permite cargar los codigos sin necesidad de poner la raspberry en bootsel mode. Además el entorno de Arduino tiene un monitor serial integral a diferencia de Pico - Visual Studio. Sin embargo tanto polling en C como polling en Arduino son muy similares en dificultad de implementación, Arduino es un poco mejor ppor unas cuantas cosas extras.
2. **MicroPython con polling: Calificación: 7** MicroPython simplifica la programación en dispositivos embebidos con Python, lo que lo hace un lenguaje intuitivo. Sin embargo, puede ser menos eficiente en términos de rendimiento comparado con C y el manejo de polling no es tan robusto como en C.
3. **C con polling: Calificación: 7** La programación en C ofrece control y eficiencia pero tiene una curva de aprendizaje un poco más complicada porque es programación a bajo nivel. El polling en C requiere una gestión cuidadosa de la memoria y el rendimiento.
4. **C con interrupciones: Calificación: 4** Razón: Aunque las interrupciones permiten una gestión eficiente y efectiva de los recursos, configurarlas correctamente en C puede ser desafiante y menos intuitivo para los principiantes o incluso para programadores intermedios.
5. **C con polling más interrupciones: Calificación: 3** Ofrece una gran eficiencia y capacidad de respuesta. Sin embargo, la complejidad de gestionar tanto polling como interrupciones en C puede hacer este enfoque uno de los más difíciles de implementar correctamente, especialmente en términos de sincronización y gestión de recursos.

Para concluir, el más complicado fue el de interrupciones y el de polling más interrupciones porque es un flujo de diseño menos familiar y requiere de la creación de las subrutinas de atención a cada interrupción, así como liberar la interrupción posterior a su uso. Incluso requiere reestructurar la lógica de las funciones encargadas de manejar los inputs. Adicionalmente, el SDK tiene una desventaja de manera práctica respecto a Micropython y Arduino, los cuales son más directos y menos engorrosos de utilizar.

0.0.2. Frecuencia máxima

La frecuencia máxima que se puede alcanzar con cada forma de onda en cada una de las 5 implementaciones son las siguientes:

1. **Frecuencia máxima con Python:** La frecuencia máxima que se puede alcanzar con Python fue de 1kHz
2. **Frecuencia máxima con Arduino:** La frecuencia máxima que se puede alcanzar con Arduino fue de 10KHz
3. **Frecuencia máxima con C:** La frecuencia máxima que se puede alcanzar con C fue de 10KHz. En una de las mejores visualizaciones se llegó a 16KHz con la implementación de Polling con C.

0.0.3. Tamaño del programa y el uso de memoria RAM en cada flujo de diseño

1. **Arduino con polling:** El tamaño es de 14kB. Uso estimado de RAM 2 kB
2. **MicroPython con polling:** El tamaño es de 7kB. Uso estimado de RAM 8 kB

3. **C con polling:** El tamaño es de 5,99 MB. Uso estimado de RAM 1.5 kB
4. **C con interrupciones:** El tamaño es de 6,73 MB. Uso estimado de RAM: 2 kB
5. **C con polling más interrupciones:** El tamaño es de 7,92 MB. Uso estimado de RAM: 3 kB

0.0.4. Discusión

Las diferencias en facilidad de programación, capacidad de generación de frecuencia y uso de recursos resaltan la importancia de seleccionar el entorno y flujo de diseño adecuados según las necesidades de cada proyecto. Mientras que Arduino y MicroPython ofrecen facilidad y rapidez de desarrollo, las implementaciones en C proporcionan mayor control, eficiencia y capacidad para manejar tareas de alta frecuencia, a cambio de una mayor complejidad y requisitos.

La implementación final del proyecto, que incluye todos los códigos fuente, documentación, procesos los cuales son necesarios para comprender y replicar el laboratorio número 3 de la materia electrónica digital 3 desarrollado para el Raspberry Pi Pico en C, está disponible públicamente para consulta y descarga en GitHub. Este repositorio representa el trabajo en equipo y las iteraciones de desarrollo que han caracterizado este proyecto desde su inicio. El link al repositorio es el siguiente: GitHub Clash Of The Titans

Conclusiones

- Es importante instalar las herramientas de desarrollo necesarias para llevar a cabo la práctica lo más pronto posible, debido a que eso causa que haya una demora para la implementación.
- Revisar la teoría de polling, interrupciones y polling más interrupciones debido a que eso hace permite que exista una mayor compresión previo a la implementación.
- Realizar algunas simulaciones previas al montaje debido a que eso reduce el daño de algunos componentes electrónicos.
- Arduino y MicroPython son accesibles y fáciles de usar, por ende son opciones ideales para prototipos rápidos y proyectos educativos. Sin embargo, esta facilidad viene con el costo de un rendimiento reducido y limitaciones en el manejo de tareas de alta complejidad.
- Las implementaciones en C, especialmente con interrupciones, ofrecen mayores capacidades de frecuencia, teniendo en cuenta la importancia de este enfoque para aplicaciones donde la respuesta en tiempo real es crucial. A pesar de su complejidad y curva de aprendizaje más compleja, C proporciona un mayor control sobre el hardware, lo que resulta en una eficiencia superior.
- En términos de tamaño del programa y uso de memoria RAM, las implementaciones en C han mostrado un uso más intensivo de espacio de almacenamiento pero han mantuvieron un uso eficiente de la RAM. Aunque requiere más código para manejar la funcionalidad de bajo nivel, optimiza el uso de recursos en tiempo de ejecución.
- Aunque las plataformas de alto nivel como Arduino y MicroPython son adecuadas para ciertos tipos de proyectos, la flexibilidad y la potencia de las implementaciones en C aseguran que sean más adecuadas para aplicaciones industriales y sistemas embebidos que exigen altos estándares de rendimiento y fiabilidad.

Referencias

[**1**] Monk, S. (2014). Programming Arduino: Getting Started with Sketches (2ª ed.). McGraw-Hill Education. [**2**] Mazidi, M. A., Mazidi, J. G., McKinlay, R. D. (2010). The AVR Microcontroller and Embedded Systems: Using Assembly and C. Pearson Education. [**3**] Horowitz, P., Hill, W. (2015). The Art of Electronics (3ª ed.). Cambridge University Press. [**4**] Williams, T. (s. f.). The Circuit Designer's Companion. [**5**]Catsoulis, J. (2005). Designing Embedded Hardware. O'Reilly Media.