INFORME DE LABORATORIO 4

Autores: *María Del Mar Arbeláez Sandoval, Julián Mauricio Sánchez Ceballos*

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

Resumen

Este proyecto utiliza una Raspberry Pi Pico para implementar un medidor del ciclo de trabajo (duty cycle) de una señal de entrada PWM con diferentes flujos de diseño: Polling, Interrupciones y Polling con interrupciones utilizando el SDK de la Raspberry Pi Pico. Usando MicroPython, el sistema monitorea la señal mediante machine.Pin y la librería time. Para Arduino se emplea el core Arduino Mbed para RP2040.

El polling revisa continuamente el estado de la señal, las interrupciones reaccionan solo ante cambios específicos de hardware, y el método combinado optimiza eficiencia y precisión. Comparando estos métodos, el proyecto permite identificar el flujo de diseño más adecuado para diferentes aplicaciones en términos de respuesta y uso de recursos.

Palabras clave: ciclo de dureza, frecuencia, flujos de desarrollo, polling, interrupciones, Arduino, MicroPython, SDK.

INTRODUCCIÓN

Este proyecto explora la implementación de un medidor de ciclo de dureza en la Raspberry Pi Pico, empleando diversos enfoques de diseño para medir el ciclo de trabajo (duty cycle) de una señal.

Utilizando los flujos de diseño polling, interrupciones y polling con interrupciones, se programa el medidor en el SDK de Raspberry Pi, Arduino y MicroPython para evaluar la eficiencia y precisión de cada método. El objetivo es comparar estos enfoques en términos de consumo de recursos y capacidad de respuesta, determinando así el método más adecuado para medir señales en aplicaciones que requieren distintos niveles de rendimiento y eficiencia.

ESQUEMÁTICO

Para este proyecto, se tiene una sección de multiplexado para los 3 displays de 7 segmentos, para que se vaya alternando un display encendido entre los tres, tan rápido que no es distinguible para los ojos. Se utiliza la siguiente imagen de referencia:

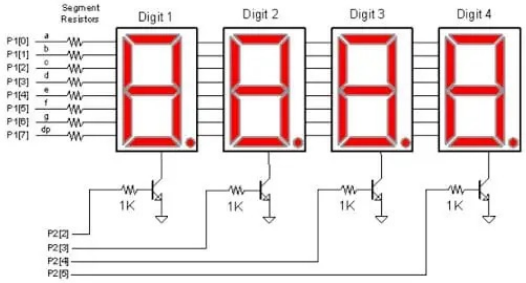

Figura **0-1**: Multiplexado de los displays de siete segmentos. Tomado de: https://controlautomaticoeducacion.com/sis embebidos/arduino/multiplexar-display-7-segmentos/

Claramente, en vez de 4 displays, se utilizarán 3, conectando los segmentos en orden A-DP de los GPIOs 2 a 9. El enable del display 1 se conecta al GPIO12, el enable del display 2 al GPIO11, el enable del display 3 al GPIO 10. Las resistencias de los segmentos son de 330Ω.

Adicionalmente, el pin de entrada del PWM es el GPIO 13, en este caso, forma parte del slice 6 y canal B del módulo PWM, debido a que en algunas implementaciones se usa este módulo como entrada.

SOFTWARE

Polling y Arduino

Estas implementaciones siguen el siguiente flujo de diseño:

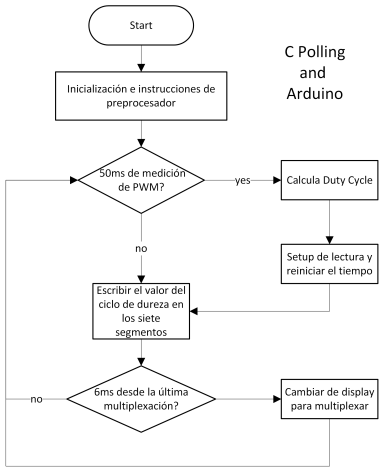

Figura **0-2**: Diagrama de flujo de Polling.

Como ambas usaron las mismas funciones y flujos, con las ligeras diferencias entre el `gpio_put_masked()` y `digital_write()`, donde se tuvo que hace un ciclo `for`.

Visualización de 7 segmentos

En esta sección se describe la implementación de la librería para la visualización de los displays de siete segmentos en un sistema embebido utilizando la Raspberry Pi Pico.

- **Inicialización de los displays**
  La inicialización de los pines asociados a los displays de siete segmentos se realiza mediante la función `init_seg()`, que configura los pines GPIO como salidas.

```
basicstyle
void init_7_seg(){
    gpio_init_mask(SEGMENTS_MASK);
    gpio_set_dir_out_masked(SEGMENTS_MASK);

    gpio_init(EN_1);
    gpio_set_dir(EN_1,true);
    gpio_init(EN_2);
    gpio_set_dir(EN_2,true);

    gpio_init(EN_3);
    gpio_set_dir(EN_3,true);
}
```

- **Escribir en los siete segmentos**
  En este fragmento solo se hace una conversión de un número a la salida de la Raspberry Pi Pico, teniendo en cuenta un booleano para la salida del punto.

```
basicstyle
/**
 * @brief Tabla de conversi n de valores a 7 segmentos
 */
uint8_t lookup[10] = {
    0x3f, 0x06, 0x5b, 0x4f, 0x66,
    0x6d, 0x7d, 0x07, 0x7f, 0x6f
};

static inline void write_value(uint8_t value, bool dp){
    gpio_put_masked(SEGMENTS_MASK,(lookup[value] | dp
        <<7)<<START_PIN);
}
```

- **Multiplexación de los displays** La función `write_decimals(uint16_t value)` se encarga de calcular el valor a mostrar en cada display según el número completo y el display que esté multiplexando, después revisa si ya se lleva más de 6ms en un display y se cambia.

```
basicstyle
void write_decimals(uint16_t value){
    static uint32_t start;
    static uint8_t en;
    uint8_t enables[]={EN_1,EN_2,EN_3,EN_1};
    uint8_t val_2_wr;

    if (value / 1000){
        val_2_wr= en==1 ? 1 : 0; //si es las cents, es 0
        if((time_us_32()-start)>6000){
            gpio_put(enables[en],false);
            write_value(val_2_wr, 0); //00,01,10 -> solo
                es true cuando es 1.
            gpio_put(enables[en+1],true);
            start=time_us_32();
            en++;
            if(en>2){en=0;}
        }
    }
    else{
        switch (en)
        {
        case 2: //units
            val_2_wr=value % 10;
            break;
        case 0: //decimals <- este va con dp
            val_2_wr=(value%100)/10;
            break;
        case 1: //cents
            val_2_wr=value/100;
            break;
        }
        if((time_us_32()-start)>6000){
            gpio_put(enables[en],false);
            write_value(val_2_wr, !en); //00,01,10 ->
                solo es true cuando es 1.
            gpio_put(enables[en+1],true);
            start=time_us_32();
            en++;
            if(en>2){en=0;}
        }
    }
}
```

Medición del duty

Esta sección se encarga de describir la implementación de la librería para la lectura del ciclo de dureza usando polling en un sistema embebido utilizando la Raspberry Pi Pico.

- **Inicialización de la detección del PWM** La inicialización del pin asociado a la lectura del PWM se realiza mediante la función `init_pwm_detect()`, que encuentra el canal (este es solo en arduino) y el slice asociado a este pin y configura el canal para la lectura.

```
basicstyle
void init_pwm_detect(){
    // Only the PWM B pins can be used as inputs.
    slice_num = pwm_gpio_to_slice_num(PWM_PIN);
    setup_duty_cycle_read(); //aqu  configura y prende
         el slice
}
```

- **Set-up del canal de PWM** Aquí, se genera la división de 100 para establecer el tick en un rango adecuado y se especifica que se tiene que contar cuando la señal de entrada esté en alto. Se le hace en enable a la señal.

```
basicstyle
void setup_duty_cycle_read(){
    // Count once for every 100 cycles the PWM B input
         is high --> Freq max 1.25M
    pwm_config cfg = pwm_get_default_config();
    pwm_config_set_clkdiv_mode(&cfg, PWM_DIV_B_HIGH);
    pwm_config_set_clkdiv(&cfg, 100);
    pwm_init(slice_num, &cfg, false);
    gpio_set_function(PWM_PIN, GPIO_FUNC_PWM);
    pwm_set_enabled(slice_num, true);
}
```

- **Calcular ciclo de dureza** En esta función se hace la división entre el contador de tiempo en alto del módulo de hardware de PWM y el máximo posible de conteo para calcular el ciclo de dureza, este resultado se multiplica por 1000 para que se tenga la respuesta en un porcentaje con un decimal.

```
basicstyle
void calculate_duty(uint16_t* duty){
    pwm_set_enabled(slice_num, false); //apaga el slice
    uint32_t count=pwm_get_counter(slice_num); // 52m se
         acaba el contador. 16 bits-> 65536
    uint32_t max_possible_count = SYS_CLK_KHZ /2;
    *duty=1000*count/max_possible_count; //se multiplica
         por 1000 para que tenga espacio para el
        decimal
    printf("max_possible count 1:%d\n",
         max_possible_count);
    printf("max_possible count 1:%d\n",
         max_possible_count);
    printf("duty count:%d\n",count);
    printf("duty percent:%d",*duty);
}
```

- **Flujo del cálculo del duty cycle** Esta función es la que retorna al main el entero de 16bits que indica el ciclo de dureza multiplicado por diez, donde se hace una espera para que se haga la lectura y luego se calcula el ciclo de trabajo, se vuelve a hacer el set-up y se retorna el valor.

```
basicstyle
uint16_t measure_duty_cycle() {
    static uint16_t duty;
    static uint32_t start;
    if((time_us_64()-start)>=50000){ //espera de 50ms
        calculate_duty(&duty);
        setup_duty_cycle_read(); //aqu  configura y
             prende el slice
        start=time_us_32();
    }
    return duty;
}
```

Interrupciones y Micropython

Para cumplir con un flujo de diseño de interrupciones se debe esperar siempre a una interrupción para hacer las operaciones necesarias. Como se muestra en la siguiente figura:

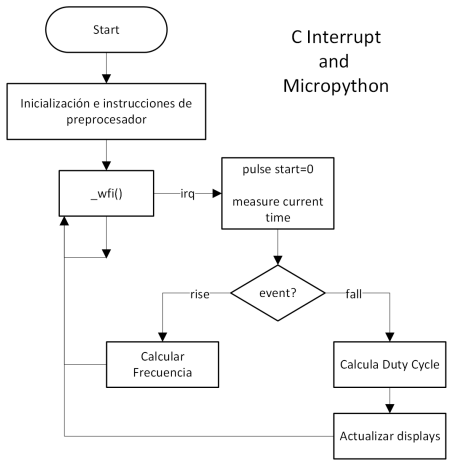

Figura **0-3**: Diagrama de flujo de Interrupciones.

Visualización de 7 segmentos

Esta sección se encarga de describir la implementación de la librería para la visualización en los displays de 7 segmentos del ciclo de dureza usando interrupciones en un sistema embebido utilizando la Raspberry Pi Pico.

Como las funciones utilizadas para el flujo de interrupciones en el SDK son las mismas que en polling en lo que se refiere a la multiplexación de los displays, no se documentará esta sección. Por otro lado, es de notar que para Micropython se usa una versión pasada un poco más simple que lo implementa sin el uso de decimales.

Medición del duty

Esta sección se encarga de describir la implementación de la librería para la lectura del ciclo de dureza usando interrupciones.

- **Inicialización de la detección del PWM** La inicialización del pin asociado a la lectura del PWM se realiza mediante la función `init_pwm_detection()`, que declara el pin como entrada, le pone un pull-down, lo enlaza con un callback y genera una detección en flancos de subida y de bajada.

```
basicstyle
void init_pwm_detection(){
    gpio_init(PWM_PIN);
    gpio_set_dir(PWM_PIN, GPIO_IN);
    gpio_pull_down(PWM_PIN);
    gpio_set_irq_enabled_with_callback(PWM_PIN,
         GPIO_IRQ_EDGE_RISE|GPIO_IRQ_EDGE_FALL, true, &
         pwm_detect_callback);
}
```

- **Callback de la detección del PWM** La función `pwm_detect_callback` calcula el período, frecuencia y ciclo de trabajo (duty cycle) de una señal PWM al detectar flancos de subida y bajada en un pin GPIO. Cuando detecta un flanco de subida, guarda el tiempo actual para calcular el período y la frecuencia. Al detectar un flanco de bajada, calcula la duración del pulso y el duty cycle como porcentaje.

```
basicstyle
void pwm_detect_callback (uint gpio, uint32_t events){
    switch (events)
    {
    case GPIO_IRQ_EDGE_RISE:
        // Guarda el tiempo actual cuando ocurre el
             flanco de subida
        pulse_time_start = 0;
        g_current_time = time_us_64();
        if (g_last_edge_time != 0){
            g_period = g_current_time - g_last_edge_time
                 ;
            g_frequency = 1000000 / (uint32_t)g_period;
        }
        g_last_edge_time = g_current_time;
        pulse_time_start = g_current_time;  // Guarda el
             tiempo de inicio del pulso
        break;

    case GPIO_IRQ_EDGE_FALL:
        // tomar el tiempo final de la se al en alto
             para el calculo del duty cycle
        g_current_time = time_us_64();
        uint64_t pulse = g_current_time -
             pulse_time_start;  // Cambiar aqu
        if (g_period != 0){
            g_duty_cycle = (pulse  * g_frequency) /
                 10000;  // Convertir a porcentaje
            printf("Duty cycle: %llu%%\n", (uint8_t)
                 g_duty_cycle);
            write_decimals((uint16_t)g_duty_cycle);
        }
        break;
    default:
        break;
    }
```

Polling + Interrupciones

Se sigue un flujo de programa con la siguiente forma:

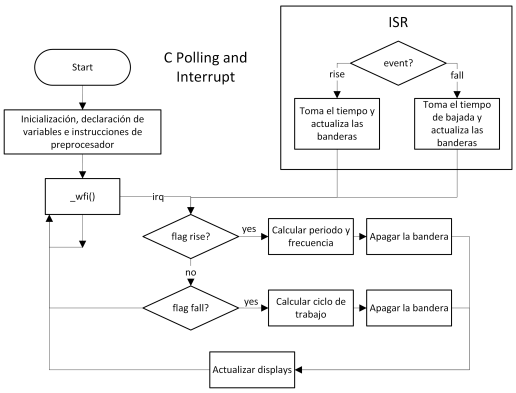

Figura **0-4**: Diagrama de flujo de Polling e Interrupciones.

Visualización de 7 segmentos

Es de notar, de nuevo, que la inicialización y la escritura de un valor en los siete segmentos se trabaja con la mismas funciones de Polling por lo que no se va a documentar en esta sección.

Medición del duty

Esta sección se encarga de describir la implementación de la librería para la lectura del ciclo de dureza usando polling con interrupciones en un sistema embebido utilizando la Raspberry Pi Pico.

- **Inicialización del pin de detección de PWM** La función `detect_pwm_init` se encarga de inicializar el pin elegido como entrada, le pone un pull-down y lo enlaza al callback, detectando flancos de subida y de bajada.

```
basicstyle
void detect_pwm_init (void ){
    gpio_init(PWM_PIN);
    gpio_set_dir(PWM_PIN, GPIO_IN);
    gpio_pull_down(PWM_PIN);

    gpio_set_irq_enabled_with_callback(PWM_PIN,
         GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true,
         &detect_pwm_callback);

}
```

- **Callback de la detección del PWM** La función `detect_pwm_callback` maneja interrupciones para detectar bordes ascendentes y descendentes en una señal PWM. Además, guarda el tiempo de cada evento y activa banderas para procesamiento con polling en el main.

```
basicstyle
void detect_pwm_callback (unsigned int gpio, uint32_t
     events){

    switch (events)
    {
    case GPIO_IRQ_EDGE_RISE:
        g_pulse_start = g_last_rise_time;
        g_last_rise_time = time_us_64();
        g_flags.edge_flag = true;
        break;

    case GPIO_IRQ_EDGE_FALL:
        g_last_fall_time = time_us_64();
        g_flags.fall_flag = true;
        break;

    default:
        break;
    }
}
```

Design Flow	Freq. Máx		Freq. Mín	
	20 %	**80 %**	**20 %**	**80 %**
Polling	$10M$	$10M$	60	60
IRQs	$15k$	$6k$	50	100
Polling + IRQs	$22k$	$15k$	25	25
Arduino	$10M$	$10M$	75	60
MicroPython	$2k$	$3k$	1	1

Cuadro **0-2**: Comparativa de frecuencias máximas y mínimas en Hz entre flujos de diseño

COMPARACIÓN ENTRE FLUJOS DE DISEÑO

En primer lugar, se hizo una comparación en dificultad de programación entre los flujos de programación y los diferentes entornos, y después se comparó el tamaño del programa:

Design Flow	Dificultad	RAM (B)
Polling	2	$1,2k$
IRQs	5	$4,4k$
Polling + IRQs	3	$4,4k$
Arduino	2	$42k$
MicroPython	10	$6,8k$

Cuadro **0-1**: Comparativa de dificultad y tamaño de memoria entre flujos de diseño

Por la parte de la dificultad, se puede ver que Polling y Arduino fueron los más simples de usar (Mayormente por la familiaridad de los entornos), el flujo con Interrupciones fue el más difícil de programar entre los flujos que se implementaron con el SDK, y Micropython, por la poca familiaridad que se maneja con el esta manera de programar el microcontrolador, fue el más difícil de programar y hacer funcionar.

En lo que se refiere a espacio en memoria RAM, se nota que el IDE de Arduino gasta sustancialmente más espacio que cualquier otro, mientras que el de polling es el que gasta menos. Es de notar que si bien Micropython gasta una buena cantidad de memoria, no es tanto para lo que se esperaría de un entorno diferente y de un lenguaje interpretado.

Finalmente, se comparan las frecuencias máximas y mínimas de los códigos, notando que el código de Polling es sustancialmente más robusto para frecuencias mayores, mientras que los códigos de Micropython y de Polling con Interrupciones son mucho mejores para frecuencias menores. Entre interrupciones y polling con interrupciones, polling parece funcionar hasta frecuencias más altas, pero no es muy diferente.

CONCLUSIONES

- Los flujos de desarrollo que utilizaron Polling y Arduino resultaron ser los más simples de implementar debido a la familiaridad con sus entornos, mientras que el uso de interrupciones y Micropython presentaron mayores desafíos, en especial este último, debido a menor familiaridad y a su naturaleza interpretada.

- El entorno de desarrollo de Arduino consume significativamente más memoria RAM en comparación con otros métodos. Polling demostró ser el método más eficiente en cuanto a consumo de memoria, mientras que Micropython, aunque también utiliza más memoria que Polling, resulta más económico de lo esperado para un lenguaje interpretado.

- Polling mostró una mayor robustez a frecuencias elevadas, siendo más adecuado para ciclos de dureza altos, posiblemente porque a diferencia del resto de flujos, se utilizó el modulo de hardware para hacer la medición. Por otro lado, los flujos de Micropython y Polling con interrupciones fueron más eficaces en frecuencias bajas, debido a que no requieren un contador que genera overflow.

BIBLIOGRAFÍA:

- Raspberry Pi Foundation. Raspberry Pi Pico documentation. Recuperado de https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html#raspberry-pi-pico-and-pico-h