

RTS Lab 1: Latencias

Yonathan López Mejía

Dpto. Ingeniería Electrónica y Telecomunicaciones
Universidad de Antioquia
Medellín, Colombia
harley.lopez@udea.edu.co

Manuel Santiago Velásquez López

Dpto. Ingeniería Electrónica y Telecomunicaciones
Universidad de Antioquia
Medellín, Colombia
msantiago.velasquez@udea.edu.co

INTRODUCCIÓN

En los sistemas de tiempo real, la *latencia de interrupción* constituye una métrica crítica para garantizar el cumplimiento de las restricciones temporales impuestas por la aplicación. Por ello, su caracterización y el estudio de estrategias para reducirla representan tareas fundamentales en el diseño y la validación de este tipo de sistemas.

En esta práctica se aborda, de manera inicial, la medición de la latencia de interrupción en el microcontrolador **AT-Mega328P**, empleando interrupciones externas generadas a partir de una señal PWM o del desbordamiento de un temporizador. Asimismo, se analiza una propuesta identificada en *Stack Exchange* que permite reducir los ciclos de reloj necesarios en una aplicación crítica de tiempo real, ofreciendo una perspectiva práctica sobre la optimización de este parámetro utilizando distintas técnicas.

PREGUNTAS ORIENTADORAS

¿En qué consiste la Latencia de Interrupción y cuál es su relación con el Tiempo de Respuesta de Interrupción? Explique.

La **latencia de interrupción** se define como el intervalo de tiempo —o el número de ciclos de reloj— que transcurre desde que se genera una señal de solicitud de interrupción (IRQ) hasta que se ejecuta la primera instrucción de la rutina de servicio a la interrupción correspondiente (ISR).

En algunas referencias, esta definición se amplía para considerar otros factores de retardo:

- El instante inicial puede definirse desde la *generación física de la señal de interrupción* y no únicamente desde la detección de la solicitud por parte del núcleo del procesador.
- El instante final puede considerarse no solo como la ejecución de la primera instrucción de la ISR, sino también como un evento desencadenado por la misma, por ejemplo, la activación de una tarea en el RTOS que ha sido despertada a partir de la interrupción.

El **tiempo de respuesta de interrupción** suele emplearse como sinónimo de la latencia de interrupción. No obstante, es posible diferenciarlos siempre que se definan las diferencias entre las terminologías para evitar confusiones.

¿En qué consiste el Context Saving & Switching y cómo se relaciona con la Latencia de Interrupción?

El *Context Saving & Switching* es el procedimiento mediante el cual la CPU conserva el estado actual de ejecución antes de atender una interrupción y posteriormente lo restaura para retomar la ejecución normal del programa. Este proceso implica almacenar el contenido de los registros de propósito general, el valor del *Program Counter* (PC), los *shadow registers* y otros elementos necesarios para garantizar que, una vez finalizada la rutina de servicio a la interrupción (ISR), el procesador pueda continuar la ejecución exactamente en el punto donde fue interrumpido.

Una vez concluida la ISR, las copias almacenadas se restauran, lo que permite reanudar el flujo normal del programa sin pérdida de información ni alteración del estado previo.

Este procedimiento ocurre en el intervalo comprendido entre la generación de la interrupción y su atención efectiva, por lo que tiene un impacto directo en la **latencia de interrupción**. Dado que las operaciones de guardado y restauración del contexto requieren varios ciclos de reloj, la latencia puede incrementarse de manera significativa. Además, en sistemas donde se producen interrupciones de forma frecuente, el costo de este proceso se acumula, afectando el rendimiento global del sistema.

¿Cuáles son las causas del Tiempo de Respuesta de Interrupción?

El tiempo de respuesta de una interrupción depende de diversos factores relacionados con la arquitectura del procesador, la gestión de las interrupciones y las características del sistema en ejecución. Entre las causas más representativas se encuentran:

- La mayoría de arquitecturas de procesadores **finalizan la instrucción en curso** antes de atender la interrupción, lo cual puede añadir varios ciclos de reloj si dicha instrucción es compleja.
- En algunos procesadores se requieren **instrucciones adicionales para identificar la ISR** correspondiente a cada interrupción, lo que incrementa la demora.
- El *Context Saving & Switching* introduce retrasos al guardar en la pila el estado del programa principal (registros, PC, etc.) y restaurarlo tras la ejecución de la ISR.

- Las instrucciones propias de la ISR deben ser **buscadas y decodificadas** como cualquier otra instrucción, lo que implica ciclos de procesamiento adicionales.
- La **velocidad de la memoria** en la que se almacena el código puede convertirse en un cuello de botella, ya que suele ser mucho menor que la del procesador, agregando ciclos extra durante la ejecución de la ISR.
- Dado que las interrupciones suelen manejar prioridades, una ISR puede ser **interrumpida por otra de mayor prioridad**, incluso al inicio de su ejecución, lo que aumenta la latencia efectiva.
- La **señal de interrupción debe sincronizarse con el reloj del procesador**; este proceso puede requerir varios ciclos adicionales antes de iniciar la atención.
- Cuando la interrupción proviene de un dispositivo externo, es necesario **sincronizarla primero con el reloj del periférico o del bus**, lo cual introduce un retardo adicional.
- En sistemas con **RTOS**, las interrupciones pueden ser temporalmente deshabilitadas durante el acceso a recursos críticos. Si una interrupción ocurre en este intervalo, su latencia aumentará hasta que el RTOS las reautorice.

¿En qué consisten los Interrupt Handlers y cuáles son sus características?

Un Interrupt Handler, también llamado ISR (Interrupt Service Routine), es la rutina de software que toma el control de la CPU cuando ocurre una interrupción. Su función es ejecutar las tareas necesarias para atender el evento generado por el hardware o el sistema operativo.

Características:

- Se ejecutan automáticamente cuando el hardware asociado o el sistema operativo detecta la interrupción.
- Están asociados a un nivel de prioridad de interrupción: una ISR de mayor prioridad puede interrumpir a otra de menor prioridad.
- El procesador el contexto antes de entrar en la ISR, y al finalizar, una instrucción especial de retorno permite restaurar dicho contexto y continuar con la ejecución normal.
- Deben ser cortas y eficientes, evitando operaciones bloqueantes o complejas, para reducir la latencia del sistema y no retrasar la atención de otras interrupciones.

Consulte y defina las Interrupciones por Desbordamiento del Temporizador (Timer Overflow Interrupt).

Una interrupción por desbordamiento del temporizador es aquella que se genera automáticamente cuando un contador de n -bits, asociado a un temporizador, alcanza su valor máximo y retorna a cero.

El funcionamiento básico de un temporizador consiste en un reloj de frecuencia fija y un registro contador de n -bits. En cada ciclo del reloj, el contador incrementa su valor en una unidad. Cuando el valor llega a $2^n - 1$ y se produce un incremento adicional, ocurre un desbordamiento, reiniciando el contador a cero.

Si las interrupciones por desbordamiento están habilitadas, cada vez que se produce este evento el hardware genera una señal de interrupción. Esto permite ejecutar una rutina de servicio (ISR) asociada al temporizador, con lo cual es posible realizar acciones periódicas y medir intervalos de tiempo con una precisión determinada por la frecuencia del reloj y el tamaño del contador.

MEDICIÓN NO. 1 DE LA LATENCIA DE INTERRUPCIÓN

Explicación de los elementos relevantes de los códigos, definiendo las señales de salida y de entrada relacionadas con la interrupción deseada, y con la medición de la latencia de interrupción en el dispositivo ATmega328p.

El código en el Arduino UNO R3 configura el pin indicado en el código tal que se encuentre disponible para efectuar una interrupción. Esta interrupción se configura tal que ocurra en el franco de subida, y efectúe un set en un pin digital a alto, genere una pausa, y después lo devuelva a bajo; esto hecho periódicamente. Las salidas y entradas del código son las siguientes:

- **Pin INT0 (PD2 / Digital 2):** Configurado como *entrada*, utilizado para recibir la señal de disparo de la interrupción externa.
- **Pin PB0 (Digital 8):** Configurado como *salida*, utilizado para generar un pulso en respuesta a la interrupción y medir la latencia.
- **Registro DDRB:** El *Data Direction Register B* (8 bits), donde se configura el pin PB0 como salida.
- **Registro PORTB:** El *Port B Data Register* (8 bits), utilizado para poner en alto o bajo el pin PB0.
- **Registro EIMSK:** El *External Interrupt Mask Register* (8 bits), habilita la interrupción externa INT0.
- **Registro EICRA:** El *External Interrupt Control Register A* (8 bits), define la condición de disparo de la interrupción (en este caso, flanco de subida en INT0).

Aunque el código original utiliza la función de trigger del osciloscopio para la obtención de datos, en el montaje se hace uso de la función de scope en tiempo real y asumiendo el peor caso en lo que concierne el estado del osciloscopio (que no posea funciones de memoria). Además, esto facilita la realización del montaje al no ser necesario un botón externo y un circuito anti rebote sino solamente un generador de señales cuadradas de ancho variable. Este tipo de señales se les va a referir como PWM por las siglas en inglés correspondientes a Pulse-Width Modulation.

Para la generación de señales PWM se hace uso de un ESP32 programado mediante la misma interfaz de Arduino. Los valores asignados a la frecuencia del módulo PWM, así como los valores posibles en bits, fueron asignados correspondientes a los parámetros permitidos en el datasheet del fabricante.

```

1  /*
2  * LAB Name: Arduino Interrupt Latency (Speed) Measurement
3  * Author: Khaled Magdy
4  * For More Info Visit: www.DeepBlueBedded.com
5  */
6  // ---[ Arduino Pin Manipulation Macros ]---
7  #define SET_PIN_MODE_OUTPUT(port, pin) DDR ## port |= (1 << pin)
8  #define SET_PIN_HIGH(port, pin) (PORT ## port |= (1 << pin))
9  #define SET_PIN_LOW(port, pin) ((PORT ## port) &= ~(1 << (pin)))
10 #define INT0_PIN 2
11
12 void INT0_ISR(void)
13 {
14     SET_PIN_HIGH(B, 0);
15     for (volatile uint16_t i = 0; i < 1000; i++) {
16         // time wasting loop for better visualization
17     }
18     SET_PIN_LOW(B, 0);
19 }
20
21 void setup()
22 {
23     SET_PIN_MODE_OUTPUT(B, 0); // Set Pin8 (PB0) As Output
24     pinMode(INT0_PIN, INPUT);
25     attachInterrupt(digitalPinToInterrupt(INT0_PIN), INT0_ISR, RISING);
26 }
27
28 void loop()
29 {
30 }
31
32

```

Fig. 1. Código modificado para funcionar con entrada PWM

```

1  // pwm resolution variable, refer to datasheet to configure alongside the frequency
2  #define LEDC_TIMER_12_812 12
3
4  // Frequency variable, refer to datasheet to configure alongside the timer
5  #define LEDC_BASE_FREQ 500
6
7  // pwm output pin
8  #define LED_PIN 4
9
10 void setup() {
11     // Setup timer with given frequency, resolution and attach it to a led pin with auto-selected channel
12     ledcSetup(LED_PIN, LEDC_BASE_FREQ, LEDC_TIMER_12_812);
13     ledcWrite(4, 512);
14 }
15
16 void loop() {
17     // Check if Fade_ended flag was set to true in ISR
18     delay(500);
19 }
20

```

Fig. 2. Código en Arduino para la generación de un PWM con el microcontrolador ESP32

Realice las capturas gráficas que sean necesarias, a fin de evidenciar la medición de dicha latencia. Verifique que dicha medida corresponde a la esperada en la simulación mostrada en el tutorial.

Para la realización de la prueba de latencia se emplean dos microcontroladores, un computador de propósito general, un componente resistivo, y un osciloscopio. El osciloscopio utilizado es analógico.

Respecto a la comparación de los datos esperados, se obtiene una latencia de 3.5 micro-segundos, la cual corresponde a un valor cercano al esperado en la simulación (3 micro-segundos), y cercano al obtenido en el montaje real realizado en el texto guía.

MEDICIÓN NO. 2 DE LA LATENCIA DE INTERRUPCIÓN

Explicación de los elementos más relevantes del código para generar la interrupción por desbordamiento del Timer, así como el valor del contador asociado.

El código configura el *Timer/Counter1* del ATmega328p en modo básico (*Normal mode*) y habilita la interrupción asociada a su desbordamiento. A continuación, se detallan los elementos más relevantes:

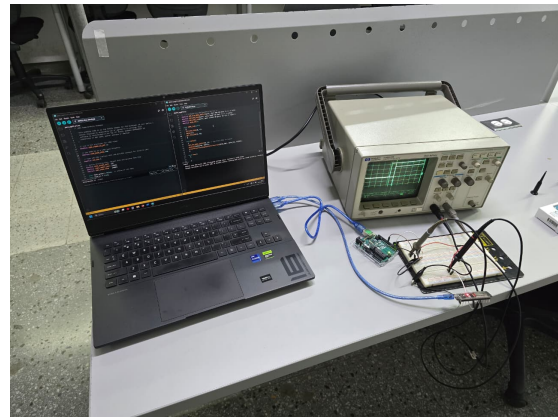


Fig. 3. Montaje físico para la realización de la prueba de latencia.

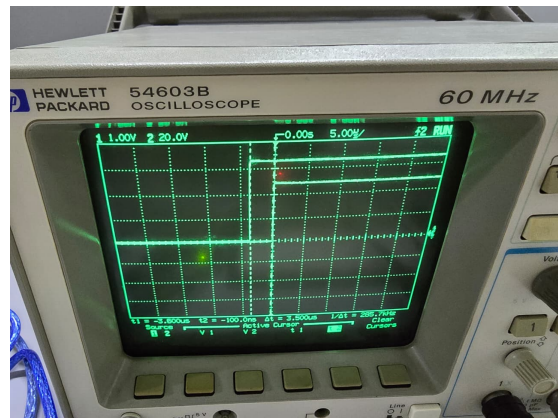


Fig. 4. Resultado de la prueba de latencia en el osciloscopio.

- **Registro TCCR1A:** El *Timer/Counter1 Control Register A* (8 bits) se asigna a cero, lo que selecciona el *Normal mode*. En este modo, el temporizador cuenta de manera ascendente hasta alcanzar su valor máximo ($2^{16} - 1 = 65535$) y posteriormente se desborda a cero. Este modo es el más simple, sin generación de PWM ni acciones automáticas sobre pines de salida.
- **Registro TCCR1B:** El *Timer/Counter1 Control Register B* (8 bits) se inicializa en cero para evitar configuraciones previas. Posteriormente, se activa el bit $CS10$ ($TCCR1B = B00000001$), lo que selecciona un preescalador de 1. Con ello, el temporizador incrementa su contador (TCNT1) a la misma frecuencia del reloj principal: $f_{clk} = 16 \text{ MHz}$. En consecuencia, el desbordamiento ocurre cada $65536/16 \text{ MHz} \approx 4.096 \text{ ms}$.
- **Registro TIMSK1:** El *Timer/Counter1 Interrupt Mask Register* (8 bits) habilita las interrupciones asociadas al temporizador. Se activa el bit $TOIE1$ (Timer Overflow Interrupt Enable), lo que permite que, al producirse un desbordamiento en TCNT1, se genere la correspondiente interrupción.
- **Comunicación Serial:** Se inicializa la UART con una velocidad de 115200 baudios para poder visualizar, desde

consola, los valores leídos durante la atención de la interrupción.

- **Rutina de servicio de interrupción (ISR):** La función `ISR(TIMER1_OVF_vect)` corresponde al manejador de interrupciones por desbordamiento del Timer1. En ella se lee el valor de `TCNT1` el cual es un contador de 16 bits y se asigna a la variable `var`. Debido a que el hardware introduce una latencia entre el instante del desbordamiento y la ejecución de la primera instrucción de la ISR, el valor leído nunca es cero, sino que corresponde a algunos ciclos posteriores. Este desfase constante permite medir experimentalmente la **latencia de interrupción**.
- La variable `var` se declara de forma global con el modificador *volatile*. Este le indica al compilador que el valor de la variable puede cambiar desde fuera del flujo normal del programa (por ejemplo, dentro de una rutina de interrupción o por acción del hardware). De esta manera se evita que el compilador aplique optimizaciones que pudieran alterar el valor leído desde ella.

Los detalles asociados a los registros se pueden consultar en la hoja de especificaciones del ATmega328P [4]

Realice las capturas de pantalla que sean necesarias a fin de evidenciar la medición de este contador. Finalmente calcule la latencia de interrupción correspondiente, y verifique que dicho valor es semejante al indicado en el tutorial estudiado.

El código programado en el Arduino se observa en la figura 1 y la salida por serial se muestra en la figura 2.

```

1  volatile int var;
2
3  ISR(TIMER1_OVF_vect) {
4      var = TCNT1;
5      Serial.println(var);
6  }
7
8  void setup() {
9      TCCR1A = 0;
10     TCCR1B = 0;
11     TCCR1B |= B00000001;
12     TIMSK1 |= B00000001;
13     Serial.begin(115200);
14 }
15
16 void loop() {
17
18 }

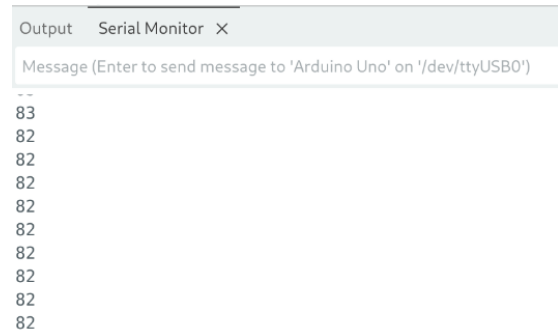
```

Fig. 5. Código para generar la interrupción de desbordamiento

Estos resultados son prácticamente idénticos a los obtenidos en el documento guía y con ellos se puede calcular la latencia de interrupción.

$$latencia = \frac{82}{16e6} = 5.13\mu s \quad (1)$$

En el artículo de referencia se indican 5 microsegundos, este valor no tiene en cuenta los decimales. Aún así, la latencia



```

Output Serial Monitor X
Message (Enter to send message to 'Arduino Uno' on '/dev/ttyUSB0')
83
82
82
82
82
82
82
82
82
82

```

Fig. 6. Salida por serial: 5 muestras que indican la cantidad de ciclos del temporizador antes de iniciar la atención de la interrupción

calculada continúa siendo cercana a aquella calculada en el numeral anterior.

LATENCIA EN UNA APLICACIÓN DE TIEMPO-CRÍTICO

En esta sección se describe la funcionalidad básica de la aplicación: un contador de flancos y de niveles altos asociados a una entrada GPIO. Esta entrada puede provenir de un pulsador, una señal PWM, o cualquier otro dispositivo externo capaz de generar una interrupción por flanco. El *firmware* emplea funciones estándar de la *libcore* de Arduino. Para leer el estado de un pin, se utiliza `digitalRead()`, mientras que para asociar una interrupción al pin se emplea `attachInterrupt()`. La ISR tiene la firma típica `void read_pin(void)`.

El programa cuenta hasta 200 interrupciones de flancos. En cada interrupción, la ISR verifica si se ha alcanzado el conteo de 200. Si no se ha alcanzado, se incrementa el contador de flancos, y si el nivel del pin es alto, también se incrementa el contador de niveles altos. Una vez que se han contabilizado las 200 interrupciones, los valores de ambos contadores se imprimen a través del puerto serie, como es habitual y se reinician los contadores.

Como entrada, se utiliza una señal PWM generada por un ESP32S3 a 100 Hz con un ciclo de trabajo del 50

Aplicación de Tiempo-Crítico - Versión Cero

Esta versión del programa es la más básica posible y utiliza todas las abstracciones de la librería de Arduino, esto facilita la escritura de código pero agrega *overhead*. A partir del análisis inicial el autor del post establece, como se lee en la tabla I, la cantidad de ciclos necesarios para leer el estado del pin 2.

La versión 0 se puede encontrar en los códigos anexos a este informe.

En el mejor de los casos la cantidad de ciclos necesarios para leer el pin son aproximadamente 99.

A continuación se observa la salida del programa:

Esta salida es plenamente correcta: al generar interrupciones por flanco se sabe que luego de uno de subida el pin estará en alto y por tanto se aumentará el contador, mientras que luego de uno de bajada el pin estará en bajo y no se realizará el conteo. Así, el contador de nivel en alto siempre deberá ser la mitad del contador de flancos.

TABLE I
RESUMEN DE CICLOS DE OPERACIÓN DE INTERRUPTOS

Operación	Ciclos
Secuencia hardwired:	≥ 4 ciclos
finalizar instrucción actual, prevenir interrupciones anidadas, guardar PC, cargar dirección del vector	
Ejecutar vector de interrupción:	3 ciclos
saltar a la ISR	
Prólogo de ISR:	32 ciclos
guardar registros	
Cuerpo principal de ISR:	13 ciclos
localizar y llamar a la función registrada por el usuario	
read_pin:	5 ciclos
llamar a digitalRead	
digitalRead:	41 ciclos
encontrar el puerto y bit relevante a probar	
digitalRead:	1 ciclo
leer el puerto de I/O	

```
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
```

Fig. 7. Salida de la aplicación para la versión 0

Para todas las demás versiones la salida será igual, el objetivo será disminuir la cantidad de ciclos necesarios.

Esta versión utiliza la forma de definir y configurar una ISR en la **Medición 1**: A un *pin* cualquiera se le asigna una interrupción de flanco utilizando `attachInterrupt()` y la ISR tiene la misma forma que la aquí utilizada.

Aplicación de Tiempo Crítico - Acceso Directo a Puertos

De la Tabla I se observa inmediatamente que la operación que más ciclos de reloj consume es la lectura del pin asociado, por lo que cualquier mejora debe centrarse en esta operación. La función `digitalRead()` es útil, pero introduce ciclos adicionales innecesarios cuando lo único que se requiere es determinar si el pin tiene un valor lógico alto (1) o bajo (0).

Dado que los periféricos están generalmente mapeados en memoria, basta con conocer qué registro corresponde al pin de interés. En este caso, los pines funcionan por defecto como entradas, y su estado se puede leer accediendo directamente al registro correspondiente mediante macros: `PIND` para los pines digitales 0 a 7, `PINC` para los pines analógicos, y `PINB` para los pines digitales 8 a 13. Cada registro tiene 8 bits, y el pin de interés corresponde al índice del bit, contando desde 0.

```
1 void read_pin()
2 {
3   uint8_t sampled_pin = PIN_REG; // leer todo el
4   registro PINx
5   if (count_edges >= MAX_COUNT) return;
6   count_edges++;
7   if (sampled_pin & (1 << PIN_NUM))
```

```
count_high++;
}
```

Listing 1. Lectura optimizada del pin mediante registro

Así, para leer el registro que contiene el valor del pin se requieren únicamente 53 ciclos frente a los 99 anteriormente necesitados.

La salida es idéntica a la fig 3

```
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
```

Fig. 8. Salida de la aplicación para la versión 1

Aplicación de Tiempo Crítico - Escribe tu propia ISR

El siguiente paso de mejora es disminuir el costo adicional de usar la función `attachInterrupt()`, esta función permite activar una interrupción, asignarle un manejador y a la vez cambiar dicho manejador en cualquier momento. Si bien este comportamiento es útil, en esta aplicación no es necesario.

Para hacerlo se requiere saber:

- El pin 2 está asociado a la interrupción externa INT0
- Se debe escribir un 1 en el bit ISC00 en el EICRA (External Interrupt Control Register A) para configurar una interrupción en cualquier flanco.
- Activar la interrupción para INT0 escribiendo un 1 en el bit INT0 del EIMSK (External Interrupt Mask Register)
- Crear la ISR para el INT0 con el macro `ISR(INT0_vect)`

Para hacer esto se debe modificar el setup y la ISR.

```
ISR(INT0_vect, ISR_NAKED)
{
  uint8_t sampled_pin = PIN_REG; // leer registro
  del puerto
  if (count_edges >= MAX_COUNT) return;
  count_edges++;
  if (sampled_pin & (1 << PIN_NUM))
    count_high++;
}

void setup()
{
  Serial.begin(9600);
  EICRA = 1 << ISC00; // detectar cualquier cambio
  en INT0
  EIMSK = 1 << INT0; // habilitar interrupcin
  externa INT0
}
```

Listing 2. Rutina de interrupción externa con acceso directo al puerto

Esta nueva aproximación disminuye los ciclos hasta aproximadamente los 20. Se debe recordar que la versión base viene de usar 99. En este punto se pierde flexibilidad para cambiar la ISR pero se gana en tiempo de ejecución. Además, como la ISR es más simple se usan menos registros.

La salida es nuevamente idéntica a las dos anteriores.

Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges

Fig. 9. Salida de la aplicacion para la version 2

Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges
Counted 100 HIGH levels for 200 edges

Fig. 10. Salida de la aplicacion para la version 3

Aplicación de Tiempo Crítico - Escribe una naked interrupt

Una interrupción tipo *naked* es aquella en que:

- El compilador no crea prólogo: No se guarda el estado del sistema automáticamente.
- El compilador no crea epílogo: No se llama a `reti()` para retornar de la ISR

Este tipo de interrupciones requieren que se cree manualmente el prólogo y el epílogo, este hecho puede aprovecharse para leer el registro al inicio del prólogo, lo que disminuye los ciclos necesarios para dicha operación. Luego, en lugar de escribir la ISR completa en *assembly* se usa una instrucción `jmp` para saltar a `ISR(INT0_vect_part_2)` y continuar con las otras instrucciones de la ISR. Teniendo en cuenta que los ciclos se están contando hasta que se lee el registro con el dato del pin, ahora solo son necesarios 10 ciclos para dicha acción.

El código modificado es el siguiente:

```
1 volatile uint8_t sampled_pin;
2
3 ISR(INT0_vect, ISR_NAKED)
4 {
5     asm volatile(
6         "___push_r0_____\n" // Guardar r0 en
           el stack
7         "___in_r0,_%[pin]_____\n" // Leer PIND en
           r0
8         "___sts_sampled_pin,_r0_____\n" // Guardar r0 en
           la variable global
9         "___pop_r0_____\n" // Recuperar r0
           desde el stack
10        "___rjmp_INT0_vect_part_2\n" // Saltar a la
           segunda parte en C
11        :: [pin] "I" (_SFR_IO_ADDR(PIND))
12    );
13 }
14
15 ISR(INT0_vect_part_2)
16 {
17     if (count_edges >= MAX_COUNT) return;
18     count_edges++;
19     if (sampled_pin & (1 << PIN_NUM))
20         count_high++;
21 }
```

Listing 3. Rutina de interrupción en dos partes: lectura temprana de PIND en ensamblador y procesamiento en C

Y la salida es idéntica a los casos anteriores
En este punto la nueva tabla de ciclos es la siguiente:

CONCLUSIONES

Se evidenció una relación inversa entre la latencia y el nivel de abstracción empleado en el desarrollo. Las interfaces de alto

TABLE II
SECUENCIA DE ATENCIÓN A UNA INTERRUPCIÓN MEJORADA

Etapas	Ciclos de reloj
Secuencia cableada (<i>hardwired</i>)	4
Vector de interrupción: salto a ISR	3
Prólogo de la ISR: salvado de registros	2
Primera instrucción en la ISR: lectura del puerto	1

nivel (APIs, ABIs, frameworks) simplifican la programación, pero introducen sobrecarga en la gestión de interrupciones. Por el contrario, las técnicas de bajo nivel —como el acceso directo a registros, la definición explícita de ISRs, el uso de ensamblador o la construcción manual de vectores de interrupción— reducen los ciclos de reloj y mejoran la latencia, aunque a costa de incrementar la complejidad del software, dificultar su mantenibilidad y aumentar la probabilidad de errores.

En consecuencia, el desarrollador debe valorar cuidadosamente en qué escenarios resulta justificable emplear dichas técnicas, ponderando el equilibrio entre desempeño y robustez del código.

En sistemas de tiempo real, donde la interacción constante con el entorno exige respuestas oportunas, la latencia de interrupción constituye un factor determinante para garantizar la correctitud temporal del sistema. Por ello, su identificación y mitigación deben considerarse objetivos centrales en el diseño de aplicaciones críticas.

Diseñar aplicaciones de tiempo críticas requieren un conocimiento robusto del *hardware* donde el *software* se ejecutará. En este ejercicio fue fundamental leer y comprender la documentación técnica del ATmega328p para escribir y entender el código propuesto.

Aunque las implementaciones sin uso de herramientas externas como osciloscopios es una forma aproximada para encontrar un valor de latencia, su precisión no es tan cercana a lo que se puede evidenciar en el primer montaje físico. Además, se puede evidenciar gran variación en lo que respecta a los tiempos de latencia medidos dependiendo de la implementación por software utilizada.

REFERENCES

[1] NXP Semiconductors, *Measuring Interrupt Latency*, application note AN12078, rev. 1, abril 2018. [En línea]. Disponible en: <https://www.nxp.com/docs/en/application-note/AN12078.pdf> [Consultado: 24-ago-2025].

- [2] DeepBlue Embedded, *Interrupt Latency & Response Time (Interrupt Speed) – Arduino*, 2025. [En línea]. Disponible en: <https://deepbluembedded.com/interrupt-latency-response-arduino/> [Consultado: 24-ago-2025].
- [3] How.dev, *Interrupt handler in OS*, 2025. [En línea]. Disponible en: <https://how.dev/answers/interrupt-handler-in-os> [Consultado: 24-ago-2025].
- [4] Microchip Technology Inc., *Atmel® ATmega328P Datasheet (Automotive Microcontrollers)*, Rev. 7810D–AVR–01/15, Microchip Technology Inc., 2015. [En línea]. Disponible en: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf [Consultado: 24-ago-2025].
- [5] avr-libc, “`<avr/interrupt.h>`: Interrupts,” *avr-libc User Manual*, Jan. 29, 2022. [En línea]. Disponible en: https://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html [Consultado: 30-ago-2025].