OPTIMIZACIÓN DE LA ADQUISICIÓN DE DATOS MEDIANTE COMUNICACIÓN UART CON DOBLE BÚFER ENTRE ARDUINO Y MATLAB.

Kevin Emmanuel Álvarez González[1] *kalvarzg1901@alumno.ipn.mx*
Linda Atenea Díaz Cruz[1] *ldiazc1800@alumno.ipn.mx*
Darinel Venegas Anaya[1] *dvenegasa@ipn.mx*
Diana Rocio Arellano Rochlin[1] *darellanor@ipn.mx*

[1]*Instituto Politécnico Nacional-Unidad Profesional Interdisciplinaria de Biotecnología.*
[2]*Instituto Politécnico Nacional - Centro de Innovación y Desarrollo Tecnológico en Cómputo.*

Resumen

Este trabajo presenta un método eficiente de adquisición de datos entre Arduino y MATLAB mediante comunicación serial asíncrona (UART). Se emplea una técnica de doble búfer con el ADC configurado en modo de ejecución libre y controlado por interrupciones, lo que permite una tasa de muestreo constante de hasta 9.6 kHz. La separación entre adquisición y transmisión de datos mejora el desempeño del sistema digital, permitiendo la visualización confiable de señales con un ancho de banda máximo de 1 kHz. Con una configuración de 115200 bps y resolución de 8 bits, el sistema resulta adecuado para aplicaciones de monitoreo

Abstract

This work presents an efficient data acquisition method between Arduino and MATLAB using asynchronous serial communication (UART). A double-buffering technique is employed with the ADC configured in free-running mode and controlled by interrupts, enabling a constant sampling rate of up to 9.6 kHz. Separating data acquisition from data transmission improves digital system performance and allows reliable visualization of signals with a maximum bandwidth of 1 kHz. Using a baud rate of 115200 bps and 8-bit resolution, the system is suitable for real-time monitoring and control applications.

I. Introducción

La exigencia de aplicaciones que requieren el monitoreo y procesamiento de señales en tiempo real, como el control avanzado o el análisis espectral, demanda sistemas de adquisición de datos con un alto grado de determinismo y elevada cantidad de datos procesados por unidad de tiempo (throughput). La implementación estándar de la lectura analógica en microcontroladores, mediante funciones que consultan constantemente el estado de un dispositivo, registro, sensor o evento para saber si ha ocurrido algo (polling) como analogRead(), introduce una latencia variable (jitter) y limita severamente la frecuencia de muestreo (fs) efectiva. A esto se suma la sobrecarga que implica el proceso de comunicación serial ineficiente y la posterior latencia de visualización en entornos de análisis como MATLAB.

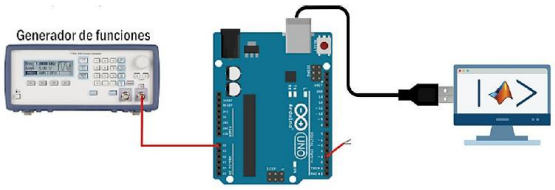


Figura 1. Esquema de conexión entre la señal de entrada (Generador de funciones), Arduino (Canal analgico A0) y Matlab (Conexión UART).

El presente trabajo aborda esta problemática proponiendo una solución integral para optimizar la cadena de adquisición de datos. Específicamente, se presenta una técnica basada en el almacenamiento de doble búfer y el control directo del Conversor Analógico-Digital (ADC) por interrupción, lo que garantiza una adquisición determinística de alta velocidad en la plataforma Arduino y una comunicación serial binaria eficiente con MATLAB. El objetivo final es establecer un enlace que permita la visualización y evaluación de señales cruciales con una latencia significativamente reducida.

II. Hardware y software utilizados

Hardware

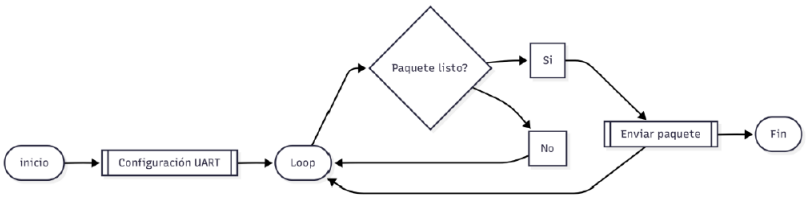
- Placa Integradora Arduino UNO/NANO (Microcontrolador ATmega328P)
- Osciloscopio
- Puntas de osciloscopio
- Generador de funciones
- Laptop/PC

Software

- IDE de Arduino (v. 1.8.19)
- Matlab (R2024b)

III. Desarrollo del código

A continuación, se presenta un análisis descriptivo del código de Arduino y el código de Matlab. (Para consultar los códigos completos visitar el repositorio de github: https://github.com/K3v1n3mm4/ComunicacionArduino-Matlab_DobleBufer.git)



Diag. de Flujo 1. Ciclo de trabajo general del código de Arduino

1. ADC en Modo de Ejecución Libre.

El modo de Ejecución Libre (Free Running) configura al ADC en un sistema de adquisición autónomo y continuo. En lugar de requerir una instrucción de software para iniciar cada conversión, el modo ejecución libre opera en un ciclo de auto disparo (auto-trigger). Este proceso se repite continuamente, estableciendo un intervalo de muestreo (Δt) estrictamente fijo determinado por el preescaler del ADC, para obtener una tasa de muestreo máxima teórica de ≈9.6 KHz, revisar Anexo A. (Microchip Technology Inc., 2016)

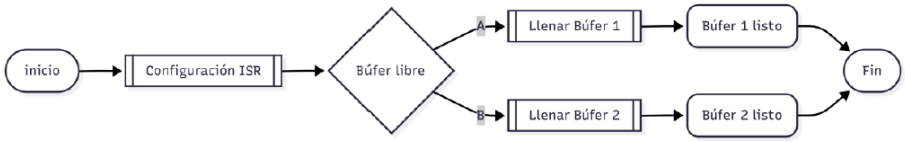
2. Rutina de Servicio de Interrupción (ISR).

Una Rutina de Servicio de Interrupción (ISR), también conocida como Interrupt Handler o Manejador de Interrupciones, es una función especial dentro del firmware de un microcontrolador que se ejecuta automáticamente y con alta prioridad cada vez que ocurre un evento de hardware específico llamado interrupción. (Microchip Technology Inc., 2016)

Conceptualmente, una ISR actúa como un mecanismo de alerta o llamado de emergencia.

- Sin ISR: El microcontrolador pasaría la mayor parte del tiempo verificando continuamente si ha ocurrido un evento (p. ej., ¿El ADC terminó? ¿Llegó un dato serial?). Esto consume tiempo de CPU y es ineficiente.
- Con ISR: Cuando el hardware detecta un evento (en este caso, la finalización de una conversión del ADC), genera una señal de interrupción. Esta señal obliga al procesador a detener inmediatamente lo que esté haciendo, guardar su estado actual y saltar a ejecutar la función ISR asociada.

Las variables compartidas entre la ISR y el código principal deben declararse como volatile esto le indica al compilador que estas variables pueden cambiar inesperadamente fuera del flujo normal del programa, asegurando que siempre se lea su valor actual. Dicha función se describe en el diagrama de flujo 2. (Microchip Technology Inc., 2016)



Diag. de Flujo 2. Lógica interna de la ISR.

3. Almacenamiento en búfer.

Un búfer es un área de memoria temporal (un colchón o amortiguador) que se utiliza para almacenar datos mientras están siendo transferidos de un lugar a otro. Ayuda a prevenir el desbordamiento (overflow) y pérdida de datos.

Doble Buffering o Ping-Pong Buffering: Consiste en usar dos áreas de memoria (Búfer A y Búfer B) que trabajan en concurrencia.

- Mientras un búfer (por ejemplo, A) está siendo llenado con nuevos datos, el otro búfer (B) está siendo vaciado (procesado o transmitido).
- Una vez que el Búfer A está lleno y el Búfer B está vacío, los roles se conmutan inmediatamente.

4. Lógica del Doble búfer y Conmutación

La gestión de los dos búferes es el mecanismo clave para la concurrencia la cual se logra usando conmutación: Cuando el búfer alcanza el límite de muestras por paquete establecido, la ISR realiza el intercambio de los búferes.

De esta forma, la ISR solo se dedica a la adquisición, mientras el lazo principal (void loop()) queda libre para gestionar la transmisión del paquete completo que se acaba de liberar. (Labrosse, 2013)

5. Protocolo de Comunicación Binaria y Transmisión Eficiente

Para la comunicación se implementó un protocolo binario, ya que la transmisión de datos como texto ASCII es ineficiente al requerir de hasta 4 o 5 bytes al intentar mandar un dato de 8 bits, (p. ej. el valor “255” requiere tres caracteres, más una coma o separador).

La función sendPacket() acepta el puntero al búfer que ha sido liberado por la ISR (buf) y realiza tres tareas secuenciales para encapsular el paquete:

$void\ sendPacket(volatile\ uint8\_t*\ buf)$ { // … $código\ de\ transmisión$ }

Y enviarlo bajo la estructura de la siguiente figura:

Figura 2. Tren de datos transmitidos y recibidos por Arduino y Matlab respectivamente.

6. Recepción y Visualización de Datos en Tiempo Real con MATLAB

El código MATLAB se encarga de establecer la comunicación serial de alta velocidad, sincronizarse con el protocolo binario enviado por Arduino, recibir el flujo de datos de la misma manera en que se envían desde Arduino mostrado en la figura 1 y realizar la visualización en tiempo real por medio de la función drawnow.

IV. Resultados

Implementando los siguientes códigos se obtienen los datos de la tabla 2. Código de Arduino:

```ino
const uint16_t HEADER = 2000; // Valor mayor a 10 bits para evitar confusión
const uint8_t NUM_SAMPLES = 100; // Tamaño del paquete

volatile uint8_t bufferA[NUM_SAMPLES];
volatile uint8_t bufferB[NUM_SAMPLES];
volatile bool useBufferA = true;
volatile uint8_t indexBuffer = 0;
volatile bool packetReady = false;

// Puntero al buffer listo para enviar
volatile uint8_t* sendBuffer = nullptr;

void setup() {
  Serial.begin(115200); // Configurar igual en Matlab

  // Configuración ADC en modo free running con interrupciones
  ADMUX = (1 << REFS0); // Referencia AVcc, canal ADC0
  ADCSRA = (1 << ADEN)  // Habilitar ADC
         | (1 << ADATE) // Auto trigger
         | (1 << ADIE)  // Interrupción habilitada
// Prescaler 64 (1 1 0) (~19.2 kHz); Modificar considerando la tabla del anexo A.
         | (1 << ADPS2) | (1 << ADPS1) | (0 << ADPS0);
  ADCSRB = 0;           // Free running
  ADCSRA |= (1 << ADSC); // Iniciar conversión
}

ISR(ADC_vect) {
  uint8_t val = ADC>>2; // Lectura de 0-1023

  if (useBufferA) {
    bufferA[indexBuffer] = val;
  } else {
    bufferB[indexBuffer] = val;
  }
  indexBuffer++;

  if (indexBuffer >= NUM_SAMPLES) {
    indexBuffer = 0;
    packetReady = true;

    // Seleccionar buffer a enviar
    if (useBufferA) {
      sendBuffer = bufferA;
      useBufferA = false;
    } else {
      sendBuffer = bufferB;
      useBufferA = true;
    }
  }
}

void loop() {
  if (packetReady) {
    sendPacket(sendBuffer);
    packetReady = false;
  }
}

void sendPacket(volatile uint8_t* buf) {
  // Enviar header (2 bytes)
 Serial.write(lowByte(HEADER));
  Serial.write(highByte(HEADER));

  // Enviar número de muestras (2 bytes)
  Serial.write(lowByte(100));

  // Enviar datos (2 bytes cada uno)
  Serial.write(lowByte(buf[i]));
}
```

Figura 3. Código de Arduino, configuración de ADC en ejecución libre, ISR y envió de paquetes

Código de Matlab:

```m
clear;
%% --- Parámetros de conexión ---
port = 'COM9';          % Cambiar por tu puerto
baud = 115200;          % Igual que Arduino baud = 230400;
HEADER = uint16(2000); % Igual que Arduino

%% --- Configurar puerto serial ---
s = serialport(port, baud);     % Cambia COM3 al puerto que uses
s.InputBufferSize = 8192;
flush(s);


%% --- Configuración de recepción ---
NUM_SAMPLES = 100;      % Igual que Arduino
HEADER_SIZE = 2;        % 2 bytes

figure;
h = plot(nan, nan);
ylim([0 260]);
xlabel('Muestra');
ylabel('Valor ADC');
title('Datos ADC desde Arduino en tiempo real');

buffer = zeros(1, NUM_SAMPLES);
while true
    % % Esperar HEADER (0xAA)
    % byte = read(s, 1, "uint8");
    % while byte ~= 170 % 0xAA
    %     byte = read(s, 1, "uint8");
    % end

    % --- Esperar HEADER (0xAA)(0x55) ---
    foundHeader = false;
    while ~foundHeader
        if s.NumBytesAvailable >= HEADER_SIZE
            rawHeader = read(s, HEADER_SIZE, 'uint8');
            valHeader = typecast(uint8(rawHeader),'uint16');
            if valHeader == HEADER
                foundHeader = true;
            end
        end
    end

    % Leer tamaño del paquete
    len = read(s, 1, "uint8");
    if len ~= NUM_SAMPLES
        continue; % descartar si tamaño no esperado
    end

    % Leer datos (2 bytes por muestra)
    data = read(s, NUM_SAMPLES, "uint8");

    % Convertir bytes a uint16 (little endian)
    % for i = 1:NUM_SAMPLES
    %     lowByte = data(i);
    %     highByte = data(2*i);
    %     buffer(i) = uint16(lowByte);
    % end

    % Graficar
    set(h, 'XData', 1:NUM_SAMPLES, 'YData', data);
    drawnow limitrate;
end
```

Figura 4. Código de Matlab, recepción y despliegue de datos.

Considerando una conversión del ADC truncada a una muestra de 8 bits y búferes de 100 muestras se puede establecer la siguiente tabla:

Tabla 1. Velocidades máximas de transmisión de datos, velocidad máxima de la frecuencia de entrada y recomendación de pre-escala ADC con relación de factibilidad.

VELOCIDAD DE COMUNICACIÓN (VCOM)		CONVERTIDOR ANALÓGICO DIGITAL (ADC)		FACTIBILIDAD	FRECUENCIA ANALÓGICA MÁXIMA SUGERIDA
[BAUD]	[bytes/s]	Pre-escala	Frecuencia de muestreo (Fs) [Ks/s]	Fs<Vcom	Fa<=Fs/10
9600	$\frac{9600}{8} = 1{,}200$	128	9.6	No	960 Hz
19200	$\frac{19200}{8} = 2{,}400$	128	9.6	No	960 Hz
38400	$\frac{38400}{8} = 4{,}800$	128	9.6	No	960 Hz
57600	$\frac{57600}{8} = 7{,}200$	128	9.6	No	960 Hz
74880	$\frac{74880}{8} = 9{,}360$	128	9.6	Posibles pérdidas de datos poco significativas	960 Hz
		64	19.2	No	1,920 Hz
115200	$\frac{115200}{8} = 14{,}400$	128	9.6	Si	960 Hz
		64	19.2	Posibles pérdidas de datos poco significativas	1,920 Hz
		32	38.4	No	3,840 Hz
230400	$\frac{230400}{8} = 28{,}800$	128	9.6	Si	960 Hz
		64	19.2	Si	1920 Hz
		32	38.4	Posibles pérdidas de datos medianamente significativas	3,840 Hz
		16	76.9	No	7690 Hz

V. Conclusiones

El método de adquisición y transmisión de datos entre Arduino-Matlab resulta útil bajo ciertas consideraciones, retomando la información de la Tabla 2., el valor mínimo de velocidad de comunicación factible para ser utilizado es 115200 baudios con la pre-escala más alta (128) siempre y cuando se requiera adquirir una señal con una resolución de 8 bits y en búferes de 100 muestras.

Velocidades de comunicación más altas, bajo las mismas condiciones de bits y número de muestras de los búfer, comenzaran a transmitir los datos más rápido de lo que se puede recibir generando un desbordamiento en los búferes lo que se traduce en despliegues dentro de Matlab acelerados de la señal; para hacer uso del método a velocidades superiores a 115200 baudios y pre escaladores más pequeños (64, 32, etc.) es necesario compensar con el número de muestras por búfer y las muestras desplegadas en Matlab (p. ej. Trabajar a 230400 a un pre escalador de 64 requiere un tamaño de búferes de 200 muestras pero al desplegar en Arduino solo muestra las primeras 100 del búfer) con la finalidad de compensar la velocidad de transmisión con la velocidad de adquisición sin afectar la visualización en tiempo real de la señal.

VI. Referencias

Arduino. (2024). Obtenido de Serial.write() — Arduino Documentation.: https://docs.arduino.cc/language-reference/en/functions/communication/serial/write/

Labrosse, J. J. (2013). Embedded systems building blocks: Complete and ready-to-use modules in C. Newnes.

MathWorks. (2024). Obtenido de typecast: https://www.mathworks.com/help/matlab/ref/typecast.html

MathWorks. (2024). Obtenido de serialport.flush documentation.: https://www.mathworks.com/help/matlab/ref/serialport.flush.html

MathWorks. (2024). Obtenido de drawnow documentation.: https://www.mathworks.com/help/matlab/ref/drawnow.html

MathWorks. (2024). Obtenido de circshift documentation.: https://www.mathworks.com/help/matlab/ref/circshift.html

Microchip Technology Inc. (2016). ATmega328P datasheet (Rev. 7810D). Obtenido de https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

VII. Anexos

Anexo A: Tabla de *pre-escalas* del ADC, valor de registros para seleccionarlo y sus características

ADPS2	*ADPS1*	*ADPS0*	*Valor binario*	*Pre-escala*	*f_ADC (Hz)*	*Tiempo conversión (µs)*	*Muestras/s*
0	0	0	000	2	8 MHz	1.625	~615 kS/s
0	0	1	001	2	8 MHz	1.625	~615 kS/s
0	1	0	010	4	4 MHz	3.25	~307 kS/s
0	1	1	011	8	2 MHz	6.5	~153 kS/s
1	0	0	100	16	1 MHz	13	~76.9 kS/s
1	0	1	101	32	500 kHz	26	~38.4 kS/s
1	1	0	110	64	250 kHz	52	~19.2 kS/s
1	1	1	111	128	125 kHz	104	~9.6 kS/s