Microcontroladores: Laboratorio 1

1st Hector Pereira
Ingeniería en Mecatrónica
Universidad Tecnológica (UTEC)
Fray Bentos, Uruguay
hector.pereira@estudiantes.utec.edu.uy

2nd Mateo Lecuna
Ingeniería en Mecatrónica
Universidad Tecnológica (UTEC)
Fray Bentos, Uruguay
mateo.lecuna@estudiantes.utec.edu.uy

3rd Mateo Sanchez
Ingeniería en Mecatrónica
Universidad Tecnológica (UTEC)
Maldonado, Uruguay
mateo.sanchez@estudiantes.utec.edu.uy

***Resumen*—**

KEYWORDS

I. Introducción

II. Marco Teórico

II-A. Microcontrolador ATmega328P

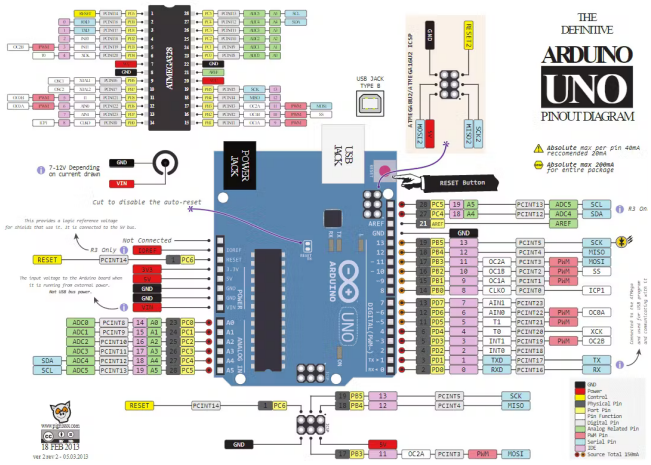

Figura 1. Diagrama de pines del Arduino Uno. Fuente: [1].

II-B. Registros de propósito general

II-C. Entradas y Salidas Digitales

- Concepto de GPIO.
- Uso de pulsadores como entradas digitales (debouncing si es necesario).
- Uso de LEDs como indicadores de estado.

II-D. Stack Pointer

El stack pointer es una herramienta utilizada dentro de la programación del microcontrolador. El stack pointer es un no es nada más que un puntero interno del microcontrolador (similar a X, Y o Z) el cual tiene una serie de operaciones asignadas, y un comportamiento específico. El stack pointer se inicializa al final de la memoria RAM, alejado de todo para intentar no molestar al resto. Allí el stack pointer se

Instrucciones como call, rcall, o las interrupciones dependen de la inicialización previa del Stack Pointer, para poder funcionar de manera adecuada, ya que el Stack Pointer es el que se encarga de guardar las direcciones de memoria a las cuales estas instrucciones deben retornar luego de terminar la ejecución, de no estar inicializado el Stack Pointer, el programa no sabría a donde volver luego de una interrupción, muy probablemente yendo a una dirección del programa aleatoria y rompiendo el flujo del codigo.

II-E. Delay activo

Un delay activo es una pieza de código que toma en cuenta el la velocidad de ejecución del microcontrolador, para ejecutar un número de instrucciones (inútiles) concreto, representando un pasaje de tiempo específico, para luego retornar al flujo del programa. Son fáciles de usar, pero no es recomendable abusard de ellas debido a ineficiencias energéticas, y que el programa se mantiene ocupado la mayor parte del tiempo en el temporizador, en lugar de poder dedicarse a hacer otras tareas.

II-F. Timers

El atmega328p tiene 3 timers diferentes: Timer 0, Timer 1, y Timer 2.

Timer 0 y 2 son de 8 bits Timer 1 es de 16 bits (puede contar más)

La ecuacion para calcular el tiempo del timer es la siguiente.

$$\frac{(2^k - C_{\text{inicio}}) \cdot \text{Prescaler}}{f_{\text{contador}}} = t_{\text{deseado}} \tag{1}$$

Esta ecuación determina el tiempo que le tomaría al contador hacer un desbordamiento (overflow) en base a la frecuencia a la cantidad de bits del contador (k), la frecuencia de trabajo del microcontrolador, el tiempo o conteo con el que se inicie el contador, y el prescaler con el que esté configurado.

II-G. Interrupciones externas

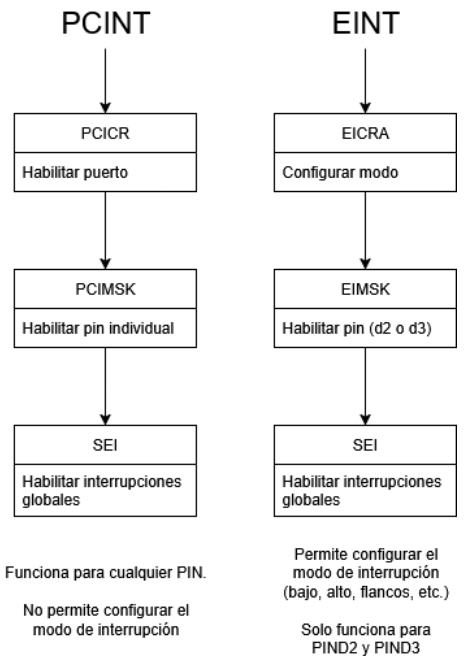

Figura 2. Flujo de configuración de interrupciones exeternas. Fuente: Elaboración própia.

II-G1. EINT:

II-G2. PCINT:

II-H. SRAM

Usando dseg se puede reservar espacio en memoria la SRAM para utilizarlo luego. Posteriormete con "variable" podremos leer o escribir el valor guardado con "lds" o "sts" respectivamente. O incluse se podría utilizar un puntero (X, Y, Z) para acceder a un lugar en memoria de manera iterativa donde quisieramos guardar múltiples valores subsecuentes (como un arreglo).

Nota: Existen 2kb de memoria RAM, el stack pointer vive dentro de la RAM así que uno debe ser precabido con el uso excesivo de este recurso.

II-I. FLASH

Utilizando .cseg y una dirección segura como 0x300 para guardar datos, se puede utilizar la misma memoria FLASH para guardar datos constantes como LUTs, fotogramas, cadenas de texto, etc. Estos datos no pueden ser modificados, pero existen 32kb de espacio en memoria FLASH para guardar información, por lo que es menos limitante que la SRAM (2kb).

Para acceder a datos guardados en program memoria del programa (FLASH) se puede hacer haciendo uso del puntero Z y la instrucción "lpm":

Se carga la dirección a Z cargando las partes bajas y altas de la dirección a ZL y ZH respectivamente. En los AVR "clásicos" (como el ATmega328P), las etiquetas en memoria de programa (.cseg) están en direcciones de palabra (cada instrucción ocupa 16 bits), pero la instrucción LPM usa una dirección en bytes en el registro Z. Por eso se hace $<< 1$ (multiplicar por 2): convierte la dirección en palabras de la etiqueta a dirección en bytes para LPM. Y "Z+" indica que luego de realizar lpm, se incremente en 1 el puntero "Z"

II-J. Bit masks

Son numeros binarios pero lo que nos importa es la posición de los unos y ceros en lugar del valor mismo que representan. Un valor binario puede representar un número o una máscara, solo depende de como lo mires.

"ldi r16, 0b00001010 out PORTB, r16" carga una máscara de bits al puerto B para encender los pines 1 y 3, y apagar el resto, pero también se puede ver como que se está cargado el número 10 al puerto B ("ldi r16, 10 out PORTB, r16")

Las máscaras son útiles cuando se busca afectar solo uno o varios de los bits de un registro. Lo que se hace primero es leer el registro, luego se realizar una operación lógica con la máscara que representa los bits que se quieren afectar, y por último se carga la máscara modificada al registro. Este tipo de operaciones se vuelve muy común cuando se trabajan con matrices LED, donde es muy común tener que mezclar puertos entre filas y columnas. Véase anexo

II-K. LUT

Para trabajar de manera ordenada en el desorden. Convierte al dolor de cabeza que son los puertos, en simples operaciones de lectura de FLASH, y modificación de punteros. Véase anexo

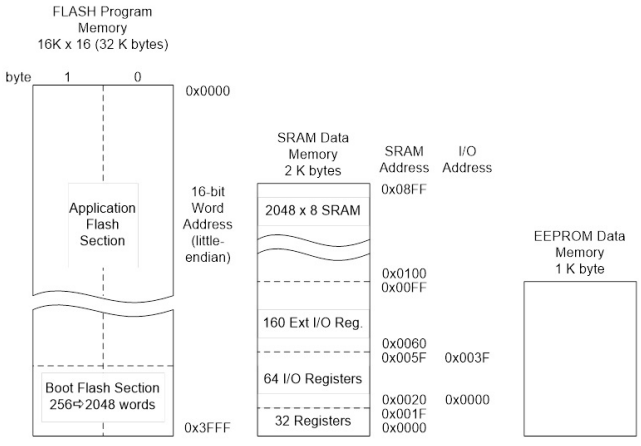

Figura 3. Mapa de memoria y espacios de direcciones en AVR de 8 bits. Fuente: [2].

II-L. USART asíncrono

II-M. Ring Buffer

II-N. USART asíncrono con Ring Buffer

II-Ñ. Automatización y Máquinas de Estado

- Qué es una máquina de estados finitos.
- Cómo se representan los estados y transiciones en un proceso automatizado (ejemplo: espera → alimentación → posicionado → punzonado → descarga → fin de ciclo).

II-O. Control de Procesos con Cinta Transportadora y Punzonadora

- Principios básicos de una cinta transportadora en automatización.
- Funcionamiento de un actuador lineal/solenoide como punzón.
- Diferentes modos de operación según carga (ligera, media, pesada).

II-P. Comunicación Serial (USART/UART)

- Definición y funcionamiento de UART.
- Ejemplos de comandos y monitoreo remoto.
- Aplicaciones en sistemas embebidos para interacción con el usuario o con PC.

II-Q. Conversión Digital-Analógica (DAC R-2R)

- Concepto de DAC y su importancia.

  Un DAC (Digital to Analog Converter) es un dispositivo o técnica que permite transformar valores digitales (códigos binarios) en señales analógicas (tensiones o corrientes continuas y variables en el tiempo). Su importancia radica en que la mayoría de los sistemas electrónicos trabajan de manera digital, pero el mundo físico es analógico: audio, imágenes, señales de control de motores, etc.
  En pocas palabras, el DAC es el "puente" entre lo digital y lo analógico.

- Arreglo de resistencias R-2R.

  El arreglo R-2R es una red de resistencias que se usa mucho para construir DACs simples y económicos. Se compone unicamente de dos valores de resistencias: una de valor R y otra de valor 2R, que se repiten en forma de escalera.
  Cada bit de nel número digital controla un interruptor (o transistor) que conecta la red a una referencia de tensión (Vref) o a tierra. Gracias a la proporción entre Ry 2R, la red genera tenciones que corresponden al valor binario aplicado.
  La ventaja del arreglo R-2R es que es fácil de implementar, no requiere de resistencias con muchos valores distintos, y mantienen una buena presición.

- Uso de una Look-Up Table (LUT) para generar señales analógicas periódicas.

  Una Look-up Table (LUT) es básicamente una tabla de valores precargada en la memoria que representa una señal digitalizada (por ejemplo, una onda seno). En lugar de calcular cada valor de la función en tiempo real, el monitor solo "lee"la tabla en orden y envía los valores a un puerto o a un DAC.
  Cuando esos valores se aplican de manera periódica y con la velocidad adecuada, en la salida se reconstruye una señal analógica periodica.

  El uso de una LUT simplifica mucho la generación de señales, por que evita cálculos complejos y garantiza que la forma de onda siempre tenga la misma calidad.

II-R. Matrices de LEDs

- Principio de funcionamiento de una matriz de LEDs.
- Multiplexado y desplazamiento de mensajes.
- Ejemplo de uso en displays.

II-S. Plotter y Control de Movimiento

- Concepto de plotter y su uso en ingeniería.
- Control de motores paso a paso o conmutados mediante relés/MOSFETs.
- Señales de control enviadas desde el microcontrolador a un PLC.

III. Metodología

III-A. Punzonadora

Maquina de estado Estados

III-B. Matriz

III-B1. Objetivo Inicial:

- Construir el control de una **matriz de LEDs 8×8 (1088AS)**.
- Empezar de forma sencilla: crear una manera de **encender un solo LED** especificando su fila y su columna.
- Expandir paso a paso hasta poder **dibujar cuadros completos** y finalmente **animaciones**.

III-B2. Encendiendo un Solo LED:

- Se crearon **lookup tables** en memoria de programa:
  - Una para los **puertos** (a qué puerto está conectada cada fila/columna).
  - Otra para los **pines** (qué número de pin dentro del puerto).
- Proceso:
  - Cargar en registros (ej. R16 y R17) la fila y la columna deseadas.
  - Usar el **puntero Z** para recorrer la tabla y determinar el puerto y el pin correctos.
  - Construir una **máscara de bits**:
    - Como se debe escribir el puerto completo, primero se toma su valor anterior.
    - Luego se aplica una máscara para encender o apagar el bit específico.
    - Se necesita manejo especial cuando filas y columnas comparten el mismo puerto.

- Nota: filas y columnas funcionan de manera inversa (la fila se activa con “0”, la columna con “1”), por lo que se usaron máscaras invertidas según corresponda.

III-B3. Multiplexado y Dibujo de Cuadros (Frames):

- Una vez resuelto el control de un LED individual, el siguiente paso fue el **multiplexado**:
  - Refrescar continuamente la matriz para que múltiples LEDs parezcan encendidos a la vez.
  - Inicialmente se implementó con un **retardo activo** (luego se consideró el uso de un temporizador).
- Se introdujeron las **tablas de frames**:
  - Los cuadros se almacenan como mapas de bits de 8×8 (0 = apagado, 1 = encendido).
  - Una subrutina de dibujo recorre las 64 posiciones:
    - Compara el bit del cuadro (usando desplazamientos como LSR).
    - Decide si debe encender el LED combinando la máscara con los puertos de hardware.
    - Operaciones lógicas aseguran que solo se actualicen los LEDs deseados.

III-B4. De Cuadros a Animación:

- Con la capacidad de renderizar un cuadro:
  - Llamar repetidamente a la subrutina de **dibujar frame** mantiene la imagen visible.
  - Ejemplo: mostrar una cara sonriente o cualquier figura estática.
- Para lograr animaciones (ej. texto desplazándose):
  - El **Timer 2** se configuró para desbordarse cada 100 ms.
  - Entre desbordes, la subrutina de dibujo corre continuamente a gran velocidad ($\approx$ cada microsegundo).
  - En cada desborde, un **puntero índice (puntero C)** se incrementa.
  - Este puntero desplaza qué parte de los datos del cuadro se lee, generando un efecto de desplazamiento en la matriz.

III-B5. Base para el Control por Estados:

- En esta etapa:
  - El sistema puede encender LEDs individuales.
  - Puede dibujar cuadros completos desde la memoria.
  - Puede ejecutar animaciones cuadro por cuadro, como texto desplazándose o patrones en movimiento.
- Esto constituye el **motor de visualización básico**, que después se conecta a la **máquina de estados** y a la **entrada por USART** (para elegir entre texto desplazante, imágenes estáticas, etc.).

III-C. Conversor

En este ejercicio se buscó visualizar en el osciloscopio la **señal 7**, correspondiente a una onda triangular.

Para ello se empleó un conversor digital-analógico (DAC) basado en una red de resistencias R-2R. El *PORTD* del microcontrolador Arduino (pines digitales 0 a 7) se conectó directamente a las entradas del DAC, permitiendo convertir los valores digitales en niveles de tensión analógicos.

En la programación del Arduino se implementó una *Look-Up Table* (LUT), en la cual se almacenaron los valores correspondientes a la forma de onda triangular. Posteriormente, estos datos se recorrieron utilizando el puntero **Z**, generando así la secuencia digital que, al pasar por el DAC, produjo la señal triangular observada en el osciloscopio.

III-D. Plotter

III-E. Materiales

1. Kit Fischertechnik
2. Cable de red
3. Resistencias variadas
4. Pulsadores
5. Matriz de leds
6. Plotter
7. Jumpers

IV. Resultados

V. Conclusiones

Referencias

[1] circuito.io. (2018) Arduino uno pinout diagram. [Online]. Available: https://www.circuito.io/blog/arduino-uno-pinout/

[2] ARXterra. Addressing modes — 8-bit avr instruction set. [Online]. Available: https://www.arxterra.com/3-addressing-modes/

[3] Microchip Technology Inc. Atmega328p datasheet. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

VI. Anexos

VI-A. Carpeta de laboratorio

Enlace de acceso a la carpeta de Google Drive con simulaciones y evidencias del laboratorio.

VI-B. Stack Pointer

Incialización del Stack Pointer:

```
RESET:
    cli ldi r16, high(RAMEND)
    out SPH, r16
    ldi r16, low(RAMEND)
    out SPL, r16 sei
    ; ...
```

Usos del Stack Pointer: rcall, call, interrupciones preservacion de datos entre subrutinas e ISRs

```
MI_ISR:
    push r16
    out r16, SREG
    push r16
```

```
    ; ...
    pop r16
    in SREG, r16
    push r16
    reti
```

VI-C. Delay activo

```
ACTIVE_DELAY:
  push r18 push r19 push r20

  ldi  r18, 1
  ldi  r19, 10
  ldi  r20, 229
L1:
  dec  r20
  brne L1
  dec  r19
  brne L1
  dec  r18
  brne L1
  nop

  pop r20 pop r19 pop r18
  ret
```

VI-D. Vectores de interrupcion

Table 11-1. Reset and Interrupt Vectors in ATmega328P

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset and watchdog system reset
2	0x002	INT0	External interrupt request 0
3	0x0004	INT1	External interrupt request 1
4	0x0006	PCINT0	Pin change interrupt request 0
5	0x0008	PCINT1	Pin change interrupt request 1
6	0x000A	PCINT2	Pin change interrupt request 2
7	0x000C	WDT	Watchdog time-out interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 compare match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 compare match B
10	0x0012	TIMER2 OVF	Timer/Counter2 overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 capture event
12	0x0016	TIMER1 COMPA	Timer/Counter1 compare match A
13	0x0018	TIMER1 COMPB	Timer/Counter1 compare match B
14	0x001A	TIMER1 OVF	Timer/Counter1 overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 compare match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 compare match B
17	0x0020	TIMER0 OVF	Timer/Counter0 overflow
18	0x0022	SPI, STC	SPI serial transfer complete
19	0x0024	USART, RX	USART Rx complete
20	0x0026	USART, UDRE	USART, data register empty
21	0x0028	USART, TX	USART, Tx complete
22	0x002A	ADC	ADC conversion complete
23	0x002C	EE READY	EEPROM ready
24	0x002E	ANALOG COMP	Analog comparator
25	0x0030	TWI	2-wire serial interface
26	0x0032	SPM READY	Store program memory ready

Figura 4. Vectores de interrupciones en el ATmega328P. Fuente: hoja de datos del ATmega328P [3].

VI-E. Timer 1

El prescaler del Temporizador 1 es configurado a través del registro TCCR1B el cual posee la siguiente configuración:

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 5. Registro de control B (TCCR1B) para Timer/Counter1. Fuente: hoja de datos del ATmega328P [3].

Los bits CS12 CS11 y CS10 configuran el prescaler del timer conforme a la siguiente tabla:

Table 15-6. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

If external pin modes are used for the Timer/Counter1, transitions on the T1 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

Figura 6. Clock Select (CS12:0) opciones de prescaler para Timer/Counter1. Fuente: hoja de datos del ATmega328P [3].

El máximo valor de tiempo que admite el Timer 1 (de 16 bits) con el prescaler de 1024 es de 4.19 segundos (aproximadamente). Para valores más grande de delay será necesario crear un contador de overflow aparte utilizando registros de uso general o SRAM

Este es un ejemplo de inicialización de timer 1 para un overflow de 1s

```
ldi r16, 0b101        sts TCCR1B, r16
ldi r16, HIGH(49911) sts TCNT1H, r16
ldi r16, LOW(49911)   sts TCNT1L, r16
```

El primer registro (TCCR1B) determina el prescaler del reloj (según la figura 6)

Utilizando la tabla de vectores de interrupcion que se muestra en la figura 4 se mapea el vector de interrupción con la etiqueta del ISR correspondiente:

```
.org 0x001A rjmp TIMER1_OVF_ISR

TIMER1_OVF_ISR:
    push r16
    out r16, SREG
    push r16
    ; ...
    pop r16
    in SREG, r16
    push r16
    reti
```

VI-F. Interrupciones Externas

VI-F1. EINT: Interrupciones externas

Table 12-1. Interrupt 1 Sense Control

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

Figura 7. Tabla de configuraciones para EICRA. Fuente: hoja de datos del ATmega328P [3].

12.2.1 EICRA – External Interrupt Control Register A

The external interrupt control register A contains control bits for interrupt sense control.

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 8. Registro de configucación de interrupciones externas EICRA. Fuente: hoja de datos del ATmega328P [3].

12.2.2 EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 9. Registro de configucación de máscaras de interrupciones externas EICRA. Fuente: hoja de datos del ATmega328P [3].

VI-F2. PCINT: Interrupciones por cambio en PIN

12.2.4 PCICR – Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 10. Registro de configucación interrupciones PCICR (PCINT). Fuente: hoja de datos del ATmega328P [3].

12.2.8 PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 11. Registro de configucación de mascara de pines para interrupciones PCIMSK0 (PCINT). Fuente: hoja de datos del ATmega328P [3].

VI-G. SRAM

```
.dseg
variable: byte 1
arreglo: byte 100
```

VI-H. FLASH

```
.cseg
.org 0x300 TABLA:
    .db 0xFF, 0x30, 0x30, 0xFF

GET_DATA:
    ldi ZH, high(TABLA<<1)
    ldi ZL, low(TABLA<<1)
    lpm r16, Z+
    ret
```

VI-I. USART Asíncrono

VI-I1. Inicialización:

```
.equ _F_CPU = 16000000
.equ _BAUD = 9600
.equ _BPS = (_F_CPU/16/_BAUD) - 1

.org 0x0024 rjmp USART_RX_ISR

RESET:
; Configurar baudios
sts UBRR0H, high(_BPS)
sts UBRR0L, low(_BPS)

; Habilitar: receptor, transmisor
; e interrupciones por RX
ldi r16, 0b10011000
sts UCSR0B,r16

; Establecer formato:
; 8 data bits, 2 stop bits
ldi r16, (1<<USBS0)|(3<<UCSZ00)
sts UCSR0C,r16

sei
```

VI-I2. Transmisión:

```
ldi r16, 0x3f ; Cargar r16
sts  UDR0, r16 ; Transmitir
```

VI-I3. Recepción:

```
USART_RX_ISR:
push r16
in r16, SREG
push r16

; Datos recibidos en r16
lds r16, UDR0
; ...

pop r16
out SREG, r16
pop r16
reti
```

VI-J. Ring Buffer

Reservando ram para el buffer

```
.dseg
tx_buffer: .byte TX_BUF_SIZE
tx_head:   .byte 1
tx_tail:   .byte 1
```

Tamaño de buffer y máscara de clamping

```
.cseg
```

```
.equ TX_BUF_SIZE = 16 ; Pot. de 2
.equ TX_BUF_MASK = TX_BUF_SIZE - 1
```

Avanzar head, y clampear con BUF MASK. Tail se puede avanzar utilizando la mísma lógica

```
lds  r17, tx_head
mov  r19, r17 ; Copiar para despues
inc  r19
andi r19, TX_BUF_MASK
```

El truco de utilizar un múltiplo de 2 y una máscara tiene la ventaja de poder limitar el rango de una variable (clamping) con tan solo una instrucción: “andi”.

Si luego de incrementar head, en el caso de que head == tail (buffer lleno), se realiza una espera activa.

```
wait_space:
; buffer lleno? next == tail
lds  r18, tx_tail
cp   r19, r18
breq wait_space
```

Guardar un valor en el buffer

```
; Z -> Cabeza de buffer
ldi  ZL, low(tx_buffer)
ldi  ZH, high(tx_buffer)
add  ZL, r17
adc  ZH, r1
st   Z, r16 ; Guardar r16 en buffer
```

En este caso se decide utilizar una espera activa, pero podría llegar a no ser una buena práctica. Otra alternativa sería simplemente descartar el dato si el buffer está lleno.

VI-K. USART asíncrono con Ring Buffer

VI-K1. Inicialización:

```
.dseg ; Ring buffer ram alocation
tx_buffer: .byte TX_BUF_SIZE
tx_head:   .byte 1
tx_tail:   .byte 1

.cseg
.equ TX_BUF_SIZE = 256
.equ TX_BUF_MASK = TX_BUF_SIZE - 1

.equ _F_CPU = 16000000
.equ _BAUD = 57600
.equ _BPS = (_F_CPU/16/_BAUD) - 1

; Recieved USART data
.org 0x0024 rjmp USART_RX_ISR
; USART Data register clear
.org 0x0026 rjmp USART_UDRE_ISR

RESET:
  clr r1

  ; Stack
  ldi r16, high(RAMEND) out SPH, r16
  ldi r16, low(RAMEND)  out SPL, r16

  ; Init USART
  ldi r16, low(_BPS)
  ldi r17, high(_BPS)
  rcall USART_INIT

  sei
  ; ...

USART_INIT:
  sts  tx_head, r1
  sts  tx_tail, r1

  sts UBRR0H, r17
  sts UBRR0L, r16

  ldi r16, 0b10011000
  sts UCSR0B,r16

  ldi r16, (1<<USBS0)|(3<<UCSZ00)
  sts UCSR0C,r16
  ret
```

VI-K2. Subrutina USART WRITE BYTE:

```
USART_WRITE_BYTE:
  push r17
  push r18
  push r19
  push ZH
  push ZL

  ; head/tail
  lds  r17, tx_head
  lds  r18, tx_tail

  ; r16 -> next = (head + 1) & MASK
  mov  r19, r17
  inc  r19
  andi r19, TX_BUF_MASK
  ; Clamping:
  ; Con 256 es 0xFF: no cambia,
  ; pero deja claro el patrón

  wait_space:
  ; buffer lleno? next == tail
  lds  r18, tx_tail
  cp   r19, r18
  breq wait_space ; espera activa

  have_space:
  ; Z -> Cabeza de buffer
  ldi  ZL, low(tx_buffer)
  ldi  ZH, high(tx_buffer)
```

```
  add  ZL, r17
  adc  ZH, r1

  st   Z, r16 ; Guardar en buffer

  ; Cabeza = next
  sts  tx_head, r19

  ; Habilitar interrupcion UDRE
  ; así el ISR comienza/continúa
  ; drenando el buffer
  cli
  lds  r18, UCSR0B
  ori  r18, (1<<UDRIE0)
  sts  UCSR0B, r18
  sei

  pop  ZL
  pop  ZH
  pop  r19
  pop  r18
  pop  r17
ret
```

VI-K3. Interrución USART UDRE ISR:

```
USART_UDRE_ISR:
  push r16
  in   r16, SREG
  push r16
  push r17
  push r18
  push r20
  push ZH
  push ZL

  ; r17 = head, r18 = tail
  lds  r17, tx_head
  lds  r18, tx_tail

  ; buffer vacío? head == tail
  cp   r17, r18
  brne usart_udre_send

  ; vacío: deshabilitar UDRIE0
  lds  r20, UCSR0B
  andi r20, ~(1<<UDRIE0)
  sts  UCSR0B, r20
  rjmp usart_udre_exit

  usart_udre_send:
  ; Z -> Cola de buffer
  ldi  ZL, low(tx_buffer)
  ldi  ZH, high(tx_buffer)
  add  ZL, r18
  adc  ZH, r1

  ; Transmitir byte
  ld   r16, Z
  sts  UDR0, r16

  ; cola = (cola + 1)
  inc  r18
  andi r18, TX_BUF_MASK
  ; Clamping:
  ; Con 256 es 0xFF: no cambia,
  ; pero deja claro el patrón

  sts  tx_tail, r18

  usart_udre_exit:
  pop  ZL
  pop  ZH
  pop  r20
  pop  r18
  pop  r17
  pop  r16
  out  SREG, r16
  pop  r16
  reti
```

VI-K4. Interrución USART RX ISR:

```
USART_RX_ISR:
  push r16
  in r16, SREG
  push r16

  ; Leer datos recibidos en r16
  lds r16, UDR0
  ; ...

  pop r16
  out SREG, r16
  pop r16
  reti
```

VI-L. Bit masks

VI-L1. SET BIT:

```
SET_BIT:
  push r16
  push r17
  push ZL
  push ZH

  ld  r17, Z     ; read current value
  or  r17, r16   ; set bit
  st  Z, r17     ; write back

  pop ZH
  pop ZL
  pop r17
```

```
  pop r16
  ret
```

VI-L2. CLEAR BIT:

```
CLEAR_BIT:
  push r16
  push r17
  push ZL
  push ZH

  ld  r17, Z    ; read current value
  com r16       ; invert mask (11110111)
  and r17, r16  ; clear bit
  st  Z, r17    ; write back

  pop ZH
  pop ZL
  pop r17
  pop r16
  ret
```

VI-M. Look Up Table (LUT)

```
; Config. de puertos filas
.org 0x310 ROW_PORTS:
  .db 0x25, 0x25, 0x25, 0x25
  .db 0x28, 0x25, 0x28, 0x28

; Config. de pines de puertos
.org 0x320 ROW_MASKS:
  .db 0b00001000, 0b00000010
  .db 0b00010000, 0b00000100
  .db 0b00001000, 0b00100000
  .db 0b00010000, 0b00100000

; Config. de puertos columnas
.org 0x330 COL_PORTS:
  .db 0x2B, 0x2B, 0x2B, 0x28
  .db 0x25, 0x28, 0x28, 0x2B

; Config. de pines de puertos
.org 0x340 COL_MASKS:
  .db 0b00010000, 0b00100000
  .db 0b10000000, 0b00000100
  .db 0b00000001, 0b00000001
  .db 0b00000010, 0b01000000
```

VI-N. Máquina de estados

```
.def current_state = r20

STATE_MACHINE:
  cpi current_state, 0
  breq STATE_MACHINE_STATE_0
  cpi current_state, 1
  breq STATE_MACHINE_STATE_1
  cpi current_state, 2
  breq STATE_MACHINE_STATE_2
  cpi current_state, 3
  rjmp STATE_MACHINE_DEFAULT

STATE_MACHINE_STATE_0:
  ; ... state logic
  rjmp STATE_MACHINE_END

STATE_MACHINE_STATE_1:
  ; ... state logic
  rjmp STATE_MACHINE_END

STATE_MACHINE_STATE_2:
  ; ... state logic
  rjmp STATE_MACHINE_END

STATE_MACHINE_DEFAULT:
  ; ... fail state logic
  rjmp STATE_MACHINE_END


STATE_MACHINE_END:
  ; ...
  ret
```