

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

El ATmega328P es un MCU AVR de 8 bits con 32 KB de memoria FLASH para programa, 2 KB de SRAM y 1 KB de EEPROM. Funciona típicamente a 16 MHz y ofrece periféricos integrados: temporizadores, USART, SPI, I²C (TWI), ADC de 10 bits e interrupciones internas y externas. Su arquitectura Harvard y las instrucciones sencillas permiten control en tiempo real con bajo consumo.

II-B. Registros de propósito general

Dispone de 32 registros de 8 bits (R0–R31) de acceso rápido. Tres pares forman punteros de 16 bits: X (R27:R26), Y (R29:R28) y Z (R31:R30), útiles para direccionamiento indirecto y acceso a tablas. El uso disciplinado de estos registros (convención de qué se preserva y qué se clobbera) facilita escribir rutinas eficientes y evitar efectos laterales.

II-C. Entradas y Salidas Digitales

Cada pin puede configurarse como entrada o salida mediante los registros DDRx; el estado lógico se escribe en PORTx y se lee en PINx. Como entrada, puede habilitarse la resistencia *pull-up* interna; como salida, el pin puede manejar LEDs, pulsadores y otros dispositivos a través de resistencias o etapas de potencia. Para pulsadores se recomienda tratamiento de rebotes; para LEDs, usar resistencias limitadoras y respetar las corrientes máximas del puerto. El mapeo de los puertos para el ATmega328P en plataforma Arduino se encuentra en el anexo ??.

II-D. Stack Pointer

El stack pointer es una herramienta utilizada dentro de la programación del microcontrolador. El stack pointer es 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. Véase anexo VI-F.

Allí el stack pointer queda como la cima de la pila y, en AVR, la pila crece hacia direcciones más bajas: un PUSH decrementa primero el SP y luego escribe el dato; un POP lee el dato y después incrementa el SP. Las llamadas a subrutinas (CALL/RCALL) empujan automáticamente la dirección de retorno (PC) a la pila y RET la recupera. En una interrupción, el hardware apila el PC y el registro de estado SREG; RETI los restaura. Además, las rutinas suelen guardar con PUSH los registros que van a usar y devolverlos con POP al salir para no corromper el contexto.

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 código. Véase anexo

II-E. Delay activo

Un delay activo es una pieza de código que toma en cuenta 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 abusar 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. Véase anexo VI-H, allí se encuentra un ejemplo de un delay activo.

II-F. Timers

Timer 0 y 2 son de 8 bits. Timer 1 es de 16 bits, por lo que puede contar intervalos más largos y con mejor resolución. Cada timer toma su reloj de la CPU y lo divide con un prescaler N . La ecuación para calcular el tiempo de desborde es:

$$\frac{(2^k - C_{\text{inicio}}) \cdot N}{f_{\text{CPU}}} = t_{\text{deseado}}$$

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. En el anexo VI-J podrá encontrar un procedimiento detallado de como configurar el Timer 1.

II-G. Interrupciones externas

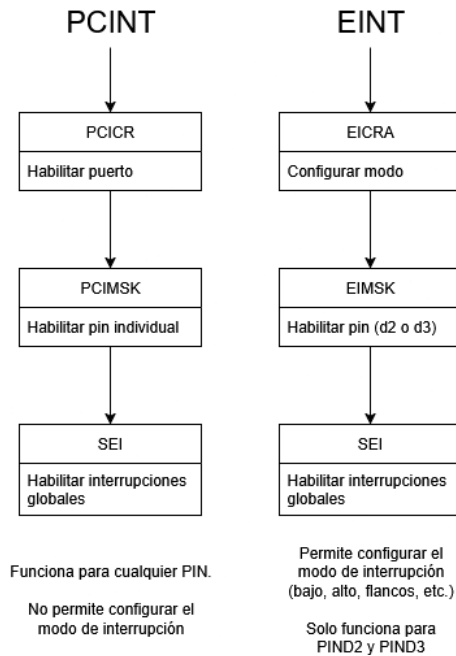


Figura 1. Flujo de configuración de interrupciones externas. Fuente: Elaboración propia.

Ver anexo VI-K para información detallada de como configurar estas interrupciones.

II-G1. EINT: Interrupciones externas en INT0 e INT1 (pines PD2 y PD3). Permiten elegir el tipo de disparo: nivel bajo, cambio lógico, flanco de bajada o de subida. Se configuran en EICRA y se habilitan en EIMSK; las rutinas de servicio son INT0_vect e INT1_vect.

II-G2. PCINT: Interrupción por cambio de pin. Detecta cualquier cambio en pines habilitados de los puertos B, C y D (grupos PCINT0_7, PCINT8_14, PCINT16_23). No distingue flancos, pero funciona en casi todos los pines. Se habilita por grupo con PCICR y por pin con PCIMSKx; las rutinas son PCINT0_vect, PCINT1_vect y PCINT2_vect.

II-H. SRAM

Usando .dseg se puede reservar espacio en SRAM. Luego, con lds/sts se lee o escribe esa variable. También pueden usarse punteros X, Y o Z para recorrer posiciones contiguas (como un arreglo). Ver anexo VI-L.

Nota: Existen 2kb de memoria RAM, el stack pointer vive dentro de la RAM así que uno debe ser precavido con el uso excesivo de este recurso, si se desea guardar gran cantidad de

valores estáticos, una mejor alternativa podrá ser la memoria FLASH.

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. En "Z" 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 incrementa en 1 el puntero Z. Véase anexo VI-M.

II-J. Bit masks

Una máscara es un número binario, pero en una máscara lo que nos importa es la posición de los unos y ceros en lugar del valor mismo que representan. Un mismo 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)

Saber operar con máscaras se vuelve útil cuando, por ejemplo, se busca afectar solo uno o varios bits en específico de un registro. En ese caso, Lo que se hace primero es leer el registro, luego se realiza 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 VI-P.

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 VI-Q

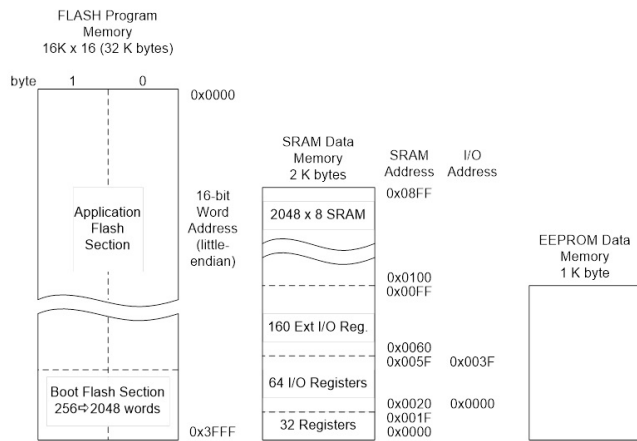


Figura 2. Mapa de memoria y espacios de direcciones en AVR de 8 bits. Fuente: [1].

II-L. USART asíncrono

La USART en modo asíncrono permite comunicación serie full-dúplex con tramas que incluyen un bit de inicio, 8 bits de datos (habitual), paridad opcional y 1-2 bits de parada. En el ATmega328P las líneas son RXD (PD0) y TXD (PD1).

La velocidad se ajusta con el registro UBRR a partir de la frecuencia de la CPU y el baud rate deseado. En modo normal (U2X0=0) se usa el reloj dividido por 16; en modo doble velocidad (U2X0=1) se usa el reloj dividido por 8, lo que mejora la precisión a ciertos baudios.

La configuración básica habilita recepción y transmisión en UCSR0B, define formato de trama en UCSR0C y consulta estados en UCSR0A; el registro de datos es UDR0. Más detalles y ejemplos en el anexo VI-N.

II-M. Ring Buffer

Un ring buffer es una cola circular que usa dos índices: *head* para escribir y *tail* para leer. Al llegar al final del arreglo, ambos vuelven al inicio (se “enroscan”), por eso es circular. La cola está vacía cuando *head* y *tail* son iguales; está llena cuando avanzar *head* una posición alcanzaría a *tail*. Elegir un tamaño potencia de dos simplifica el avance de índices (se puede usar una máscara en lugar de un módulo). En USART suele emplearse para recibir y transmitir sin bloquear: la ISR coloca bytes en la cola y el código principal los consume. Conviene proteger la actualización de índices compartidos y decidir qué hacer si se llena la cola. Más detalles en el anexo VI-Ñ.

II-N. USART asíncrono con Ring Buffer

La combinación de USART asíncrono con un ring buffer permite comunicación serie sin bloqueo. En recepción, la ISR de RX lee UDR0 y encola el byte en el buffer de entrada; la aplicación lo desencola cuando conviene. En transmisión, la aplicación encola datos en el buffer de salida y habilita la interrupción *Data Register Empty*; la ISR va sacando bytes y cargándolos en UDR0, deshabilitando la interrupción cuando la cola queda vacía. Esta arquitectura evita esperas activas y

minimiza la pérdida de datos bajo carga. Es recomendable usar tamaños potencia de dos, proteger la actualización de índices compartidos y definir una política ante desbordes (descartar lo nuevo o lo viejo, o espera activa). Detalles y esquema de inicialización en el anexo VI-O.

II-Ñ. Automatización y Máquinas de Estado

Una máquina de estados modela el proceso como un conjunto finito de estados (p. ej., espera, mover cinta, punzonar, retroceso) y transiciones disparadas por eventos (sensores, temporizadores, comandos por USART). En cada estado se definen acciones de entrada/salida (activar motores, habilitar solenoide, actualizar LEDs) y las condiciones para cambiar al siguiente estado. Este enfoque hace el control *determinista*, facilita el manejo de errores y simplifica la verificación de seguridad (interlocks, límites de carrera, anti-rebote en pulsadores). En el ATmega328P se implementa de forma directa con un “switch” sobre la variable de estado y saltos condicionales, manteniendo la lógica de transición en una sección única y clara. Los temporizadores e interrupciones proveen eventos periódicos y asincrónicos sin bloquear el ciclo principal. Un diagrama y la especificación de estados/transiciones se incluyen en el anexo VI-R.

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

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.

El arreglo R-2R es una red de resistencias que se usa mucho para construir DACs simples y económicos. Se compone únicamente de dos valores de resistencias: una de valor R y otra de valor 2R, que se repiten en forma de escalera.

Cada bit del número digital controla un pin de los puertos del microcontrolador que conecta la red a una referencia de tensión (V_{ref}) o a tierra. Gracias a la proporción entre R_y 2R, la red genera tensiones que corresponden al valor binario aplicado.

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 microcontrolador “lee” la tabla en orden y envía los valores a un puerto configurado con un DAC. Cuando esos valores se aplican de manera periódica y con la velocidad adecuada, en la salida se reconstruye una señal analógica periódica.

II-P. Matriz de LEDs

Modelo 1088AS: matriz 8x8 de cátodo/anodo multiplexado, apta para control por filas y columnas. La disposición exacta de pines se encuentra en el anexo VI-C. Para encender un LED específico, se aplica nivel 0 a la fila correspondiente y nivel 1 a la columna correspondiente.

II-Q. Plotter y Control de Movimiento

Un plotter es un sistema de trazado que posiciona un útil (lápiz/solenoides) en X-Y para dibujar o marcar piezas. En ingeniería se usa para prototipado, plantillas y pruebas de control de movimiento.

El movimiento puede lograrse con motores paso a paso o con motores conmutados mediante relés/MOSFETs. En el primer caso se controlan pasos y dirección; en el segundo, sentido y velocidad (opcionalmente con PWM). El útil sube y baja con un solenoide controlado también por una salida digital.

Este plotter ya tiene un Arduino conectado al PORTD. Cada bit de PORTD activa una acción: arriba, abajo, izquierda, derecha, subir solenoide, bajar solenoide. La tabla de comandos está en el anexo VI-D.

II-R. Macros

En ensamblador, una macro es una plantilla de texto que el ensamblador expande en tiempo de ensamblado. No es una subrutina: no hay llamadas ni RET; el código se copia inline cada vez que se usa. Sirven para evitar repetir secuencias, parametrizarlas y hacer el código más legible. Véase anexo VI-S.

Entre las ventajas están la reutilización de patrones, la reducción de errores por copia y pega, y la posibilidad de recibir parámetros (puerto, bit, conteo, etc.), lo que permite adaptar la misma secuencia a distintos casos.

Como cuidado, cada expansión aumenta el tamaño del binario. Además, como el cuerpo se inserta tal cual, conviene documentar qué registros modifica (clobbers) y usar etiquetas locales para evitar choques de nombres.

III. METODOLOGÍA

III-A. Punzonadora

Maquina de estado Estados

III-B. Matriz

III-B1. Objetivo inicial: El objetivo es controlar una **matriz de LEDs 8x8 (1088AS)**. Se comienza con lo básico: encender un solo LED indicando su fila y su columna. A partir de ahí se progresa hasta dibujar cuadros completos y finalmente animaciones.

III-B2. Encendiendo un solo LED: Para ubicar cada LED se prepararon **lookup tables** en memoria de programa. Una tabla indica el **puerto** asociado a cada fila y columna y otra el **pin** dentro de ese puerto. El flujo es simple: se cargan en registros la fila y la columna deseadas, se usa el **puntero Z** para leer puerto y pin desde las tablas y se construye una **máscara de bits**. Primero se lee el valor actual del puerto y luego se aplica la máscara para encender o apagar el bit. Cuando fila y columna comparten puerto se agrega un cuidado extra para no alterar otros bits. Recordatorio importante: las **filas** se activan con 0 y las **columnas** con 1, por lo que se emplean máscaras invertidas según corresponda.

```
ldi ZH, high(ROW_PORTS<<1)
```

```
ldi ZL, low(ROW_PORTS<<1)
add ZL, row adc ZH, r1
lpm r16, Z ; r16 = row port address
```

```
ldi ZH, high(ROW_MASKS<<1)
ldi ZL, low(ROW_MASKS<<1)
add ZL, row adc ZH, r1
lpm r17, Z ; r18 = row pin mask
```

```
; Encender fila
clr ZH mov ZL, r16
mov r16, r17
rcall CLEAR_BIT
```

```
ldi ZH, high(COL_PORTS<<1)
ldi ZL, low(COL_PORTS<<1)
add ZL, col adc ZH, r1
lpm r16, Z ; r16 = column port address
```

```
ldi ZH, high(COL_MASKS<<1)
ldi ZL, low(COL_MASKS<<1)
add ZL, col adc ZH, r1
lpm r17, Z ; r17 = column pin mask
```

```
; Encender columna
clr ZH mov ZL, r16
mov r16, r17
rcall SET_BIT
```

III-B3. Multiplexado y dibujo de cuadros: Resuelto el encendido individual, se pasa al **multiplexado** para refrescar la matriz de forma continua y dar la ilusión de múltiples LEDs encendidos. Al principio se usó un retardo activo y luego se contempló el uso de un temporizador. Los **frames** se guardan como mapas de bits 8x8 donde 0 es apagado y 1 encendido. Una subrutina recorre las 64 posiciones, compara el bit del cuadro (con desplazamientos como LSR), decide si encender el LED combinando la máscara con los puertos y aplica operaciones lógicas para modificar solo los LEDs necesarios.

```
ldi row, 0 RENDER_FRAME_ROW_LOOP:
ldi r16, 0b00000001 ; Frame mask
lpm r17, Z+
```

```
ldi col, 0 RENDER_FRAME_COL_LOOP:
rcall CLEAR_MATRIX
```

```
push r16
and r16, r17
```

```
cpi r16, 0
breq RENDER_FRAME_SKIP_LED
```

```
rcall TURN_LED
rcall TEST_DELAY
```

```

    RENDER_FRAME_SKIP_LED:
    pop r16
    lsl r16
    inc col cpi col, 8
    brlo RENDER_FRAME_COL_LOOP

    inc row cpi row, 8
    brlo RENDER_FRAME_ROW_LOOP

```

III-B4. De cuadros a animación: Con la rutina de dibujo funcionando, basta con llamarla en bucle para mantener la imagen. Así se pueden mostrar figuras estáticas como una cara sonriente. Para animaciones, por ejemplo texto desplazándose, el **Timer 2** se configura con un desborde cada 100ms. Entre desbordes la rutina de dibujo corre a alta velocidad (del orden de microsegundos). En cada desborde se incrementa un **puntero índice** que desplaza el origen de lectura de los datos del cuadro y produce el efecto de movimiento sobre la matriz. Además se agrega una condición de comparación que retorna la animación al inicio cuando se llega al final. Vease anexo VI-J

III-B5. Base para el control por estados: Con estas piezas el sistema ya enciende LEDs individuales, dibuja cuadros desde memoria y ejecuta animaciones cuadro a cuadro. Este es el **motor de visualización básico** que luego se conecta a la **máquina de estados** y a la **entrada por USART** para seleccionar entre texto desplazante, imágenes estáticas u otros patrones. Vease anexo VI-R y anexo VI-O

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

Para el plotter se tomaron las instrucciones de la tabla mostrada en el anexo VI-D

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] ARXterra. Addressing modes — 8-bit avr instruction set. [Online]. Available: <https://www.arxterra.com/3-addressing-modes/>
- [2] Microchip Technology Inc. Atmega328p datasheet. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
- [3] circuito.io. (2018) Arduino uno pinout diagram. [Online]. Available: <https://www.circuito.io/blog/arduino-uno-pinout/>
- [4] Software Particles. (2024, Apr.) Learn how an 8x8 led display works and how to control it using an arduino. Figura: 8x8 LED matrix pin mapping (imagen del artículo). [Online]. Available: <https://softwareparticles.com/learn-how-an-8x8-led-display-works-and-how-to-control-it-using-an-arduino/>

VI. ANEXOS

VI-A. Carpeta de laboratorio

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

VI-B. Microcontrolador ATmega328P

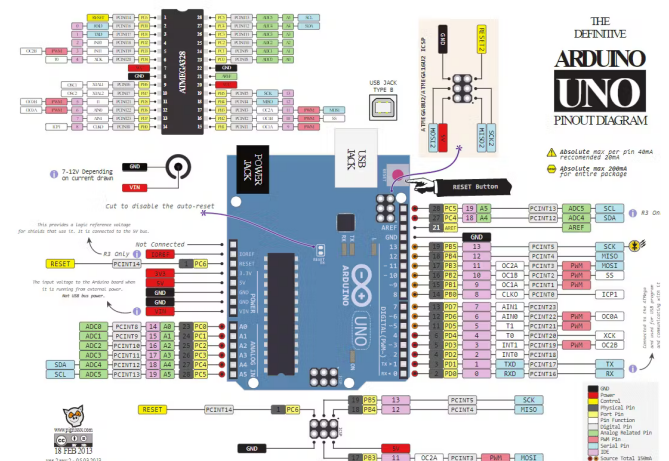


Figura 3. Diagrama de pines del Arduino Uno. Fuente: [3].

VI-C. Matriz de LEDs 1088AS

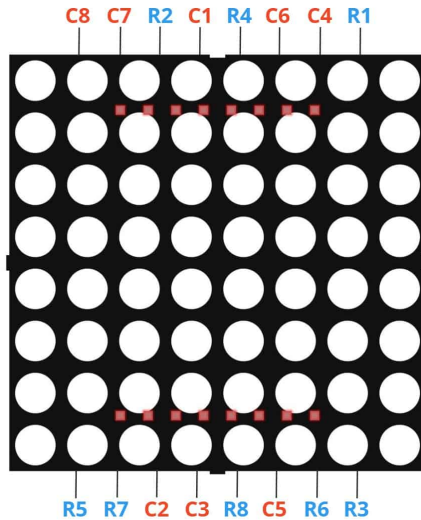


Figura 4. Pinout de Matriz de LEDs 1088AS. Fuente: [4].

VI-D. Comandos Plotter

Pin Digital	Conexión
D2	Bajar solenóide X0
D3	Subir solenóide X1
D4	Movimiento hacia abajo X5
D5	Movimiento hacia arriba X6
D6	Movimiento hacia la izquierda X7
D7	Movimiento hacia la derecha X10

Figura 5. Comandos del Plotter del laboratorio de Mecatrónica. Fuente: Hoja de laboratorio.

VI-E. Señal para conversor digital analogo

```
0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38,
0x40, 0x48, 0x50, 0x58, 0x60, 0x68, 0x70, 0x78,
0x80, 0x88, 0x90, 0x98, 0xa0, 0xa8, 0xb0, 0xb8,
0xc0, 0xc8, 0xd0, 0xd8, 0xe0, 0xe8, 0xf0, 0xf8,
0xff, 0xf7, 0xef, 0xe7, 0xdf, 0xd7, 0xcf, 0xc7,
0xbf, 0xb7, 0xaf, 0xa7, 0x9f, 0x97, 0x8f, 0x87,
0x7f, 0x77, 0x6f, 0x67, 0x5f, 0x57, 0x4f, 0x47,
0x3f, 0x37, 0x2f, 0x27, 0x1f, 0x17, 0x0f, 0x07
```

VI-F. Stack Pointer

Inicialización 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

VI-G. Interrupt service routines

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

VI-H. 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-I. Vectores de interrupción

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	0x0002	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 6. Vectores de interrupciones en el ATmega328P. Fuente: hoja de datos del ATmega328P [2].

VI-J. Configuración Timer 1

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

Bit (0x81)	7	6	5	4	3	2	1	0	
	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 7. Registro de control B (TCCR1B) para Timer/Counter1. Fuente: hoja de datos del ATmega328P [2].

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	$clk_{clk}/1$ (no prescaling)
0	1	0	$clk_{clk}/8$ (from prescaler)
0	1	1	$clk_{clk}/64$ (from prescaler)
1	0	0	$clk_{clk}/256$ (from prescaler)
1	0	1	$clk_{clk}/1024$ (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 8. Clock Select (CS12:0) opciones de prescaler para Timer/Counter1. Fuente: hoja de datos del ATmega328P [2].

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 8)

Utilizando la tabla de vectores de interrupcion que se muestra en la figura 6 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-K. Interrupciones Externas

VI-K1. 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 9. Tabla de configuraciones para EICRA. Fuente: hoja de datos del ATmega328P [2].

12.2.1 EICRA – External Interrupt Control Register A

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

Bit (0x09)	7	6	5	4	3	2	1	0	
	–	–	–	–	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 10. Registro de configuración de interrupciones externas EICRA. Fuente: hoja de datos del ATmega328P [2].

12.2.2 EIMSK – External Interrupt Mask Register

Bit (0x1D) (0x3D)	7	6	5	4	3	2	1	0	
	–	–	–	–	–	–	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 11. Registro de configuración de máscaras de interrupciones externas EICRA. Fuente: hoja de datos del ATmega328P [2].

VI-K2. PCINT: Interrupciones por cambio en PIN

12.2.4 PCICR – Pin Change Interrupt Control Register

Bit (0x0B)	7	6	5	4	3	2	1	0	
	–	–	–	–	–	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 12. Registro de configuración interrupciones PCICR (PCINT). Fuente: hoja de datos del ATmega328P [2].

12.2.8 PCMSK0 – Pin Change Mask Register 0

Bit (0x0B)	7	6	5	4	3	2	1	0	
	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 13. Registro de configuración de mascara de pines para interrupciones PCIMSK0 (PCINT). Fuente: hoja de datos del ATmega328P [2].

VI-L. SRAM

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

VI-M. 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-N. USART Asíncrono

VI-N1. 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-N2. Transmisión:

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

VI-N3. 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-Ñ. 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 misma 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-O. USART Asíncrono con Ring Buffer

VI-O1. Inicialización:

```
.dseg ; Ring buffer ram allocation
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-02. 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-03. Interrupción 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-O4. Interrupción 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-P. Bit Masks

VI-P1. 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-P2. 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-Q. Look Up Table (LUT)

```

; Config. de puertos filas
.org 0x300
ROW_PORTS:
    .db 0x25, 0x25, 0x25, 0x25
    .db 0x28, 0x25, 0x28, 0x28

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

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

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

```

VI-R. 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
```

VI-S. Macros

Definir macros en archivo “.inc”:

```
.macro DISABLE_TIMER_1
push r16
ldi r16, 0      sts TCCR1B, r16
ldi r16, (0<<TOIE1)  sts TIMSK1, r16
ldi r16, (1<<TOV1)   out TIFR1, r16
pop r16
.endmacro
```

```
.macro ENABLE_TIMER_1
; @0 Timer seconds
push r16
mov timer1_ovf_counter, @0
ldi r16, 0b101    sts TCCR1B, r16
ldi r16, HIGH(49911)  sts TCNT1H, r16
ldi r16, LOW(49911)   sts TCNT1L, r16
ldi r16, (1<<TOV1)   out TIFR1, r16
ldi r16, (1<<TOIE1)  sts TIMSK1, r16
pop r16
.endmacro
```

Importar macros a “main.asm”:

```
.include "m328pdef.inc"
.include "macros.inc"
```

VI-T. Plotter: Generador de círculos con Python

```
import math
```

```
# Number of steps per quadrant
divisions = 42
```

```
# Maximum cycles per motor
max_cycles = 100
solenoid = "SOLENOID_DOWN"
```

```
# Define quadrants with corresponding
# directions and wave signs
```

```
quadrants = [
    ("RIGHT", "UP", 1, 1),      # Q I
    ("DOWN", "RIGHT", -1, 1),  # Q II
    ("LEFT", "DOWN", -1, -1),  # Q III
    ("UP", "LEFT", 1, -1)      # Q IV
]
```

```
# Function to scale
# wave values to cycles
```

```
def get_cycles(value1,
                value2,
                max_cycles):
    max_val = max(value1, value2)
    if max_val == 0:
        return 0, 0
```

```
    scale = max_cycles / max_val
    return int(round(value1 * scale)),
    int(round(value2 * scale))
```

```
# Generate commands
```

```
for quad_index, (dir1,
                 dir2,
                 sign1,
                 sign2)
in enumerate(quadrants):
```

```
    print(f"\n; Quadrant
    {quad_index + 1}: {dir1} & {dir2}")
```

```
for step in range(divisions):
    # Calculate sine/cosine wave (0..1)
    angle = (math.pi / 2) *
    (step / (divisions - 1))
```

```
    wave1 = math.sin(angle) * sign1
    wave2 = math.cos(angle) * sign2
```

```
# Take absolute value because
# we only care about time cycles
wave1, wave2 = abs(wave1),
abs(wave2)
```

```
# Scale to cycles
cycles1, cycles2 =
get_cycles(wave1,
            wave2,
            max_cycles)
```

```
# Both motors on for min(cycles1,
```

```

#                                     cycles2)
i_cycles = min(cycles1, cycles2)
# Only the \extra" cycles
# for the motor that is ahead
j_cycles = abs(cycles1 - cycles2)

# Determine which motor
# gets the extra cycles
if cycles1 > cycles2:
    extra_motor = dir1
else:
    extra_motor = dir2

# Output commands
if i_cycles > 0:
    print(f" .db {i_cycles},
          {solenoid} + {dir1} + {dir2}")

if j_cycles > 0:
    print(f" .db {j_cycles},
          {solenoid} + {extra_motor}")

```