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*—Se presenta el diseño e implementación de cuatro subsistemas sobre el microcontrolador ATmega328P: una matriz de LEDs 8x8, un plotter cartesiano, un conversor digital-analógico y una punzonadora con cinta. Cada módulo ejercita competencias clave de sistemas embebidos: multiplexado de filas/columnas y manejo de temporizadores, interpretación de comandos y generación de trayectorias, síntesis y verificación de señales, y control secuencial por máquina de estados con disparo desde USART y pulsadores. Se validó el funcionamiento estable de los módulos, con latencia de comunicación adecuada y formas de onda coherentes con lo esperado. Como líneas de mejora se identificaron el uso de PWM para control fino de brillo y velocidad, perfiles de movimiento para el plotter e integración de sensores en la lógica de estados de la punzonadora.**

Palabras clave: ATmega328P, AVR, sistemas embebidos, matriz de LEDs, multiplexado, temporizadores, interrupciones, USART, plotter cartesiano, trayectorias, conversor digital-analógico, PWM, máquina de estados.

I. Introducción

Este laboratorio integra cuatro desarrollos coordinados —matriz de LEDs 8x8, plotter cartesiano, conversor digital–analógico y punzonadora con cinta— para ejercitar diseño, programación y validación de sistemas embebidos sobre el microcontrolador ATmega328P. A partir de una arquitectura modular, cada subsistema aborda un conjunto distinto de competencias: generación y multiplexado de patrones en la matriz, interpretación de comandos y generación de trayectorias en el plotter, síntesis de señales en el conversor y control secuencial por estados en la punzonadora. En conjunto, el trabajo permite recorrer el ciclo completo de desarrollo: especificación, implementación en C/ensamblador, pruebas instrumentadas y análisis de resultados.

Objetivos específicos

- Implementar manejo eficiente de GPIO, temporizadores, interrupciones y comunicación por USART.
- Multiplexar filas/columnas para presentar patrones en la matriz y evaluar latencia visual.
- Generar trayectorias del plotter a partir de comandos interpretados y scripts en Python.
- Producir formas de onda con el conversor y compararlas con las simulaciones previstas.
- Diseñar y validar una máquina de estados para la punzonadora, incluyendo arranque por pulsadores y por USART.

II. Marco Teórico

II-A. Microcontrolador ATmega328P

El ATmega328P es un microcontrolador 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, I^2C (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 VII-C.

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 VII-G.

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 codigo.

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 VII-I, 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:

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

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 VII-K podrá encontrar un procedimiento detallado de como configurar el Timer 1.

II-G. Interrupciones externas

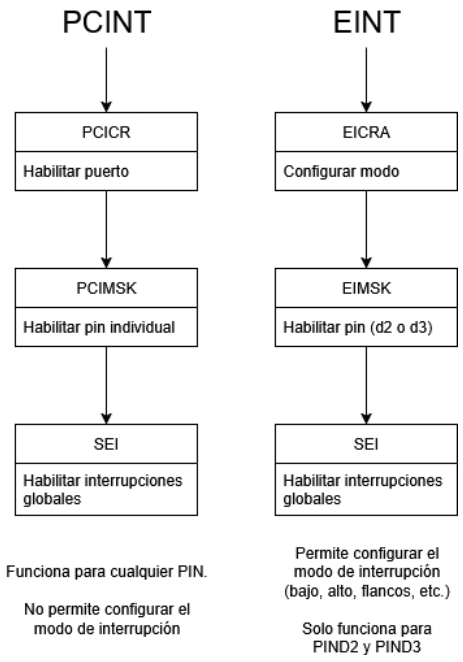

Figura 1. Flujo de configuración de interrupciones externas. Fuente: Elaboración propia.

Ver anexo VII-L 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 PCMSKx; 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 VII-M.

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 podría 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 incremente en 1 el puntero Z. Véase anexo VII-N.

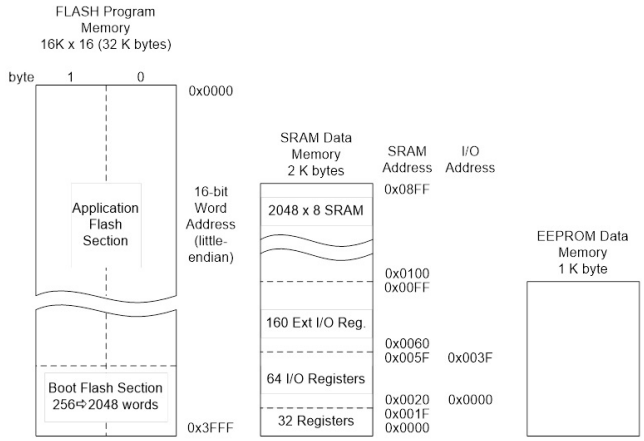

Figura 2. Mapa de memoria y espacios de direcciones en AVR de 8 bits. Fuente: [1].

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 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 VII-Q.

II-K. LUT

Una LUT o Look Up Table, permite establecer un orden o mapear información de manera que se pueda trabjar de manera simplificada utilizando punteros dentro del programa. Una LUT suele ser utilizada para mapear la distribución de puertos y pines en la conexión de un circuito con elementos posicionales como lo es una matriz de LEDs (por ejemplo). Se vuelven de gran ayuda para no depender de lógica del circuito para adaptarse a diferentes configuraciones de pines, y poder adaptar circuitos de manera mucho más símple. Véase anexo VII-R

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 VII-Ñ.

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 VII-O.

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 VII-P.

Para envíar cadenas de texto a través de USART se utiliza la función vista en el anexo VII-P5: Al precargar Z con la dirección en memoria de programa de un mensaje guardado, envía contínuamente un mensaje hasta que se encuentra con el valor 0, que le indica el final del mensaje:

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, antirebote 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 VII-S.

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 unicamente 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 (Vref) o a tierra. Gracias a la proporción entre Ry 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 8×8 de cátodo/anodo multiplexado, apta para control por filas y columnas. La disposición exacta de pines se encuentra en el anexo VII-D. 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/solenoide) 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 VII-E.

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 VII-T.

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. Materiales a utilizar

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

III-B. Punzonadora

III-B1. Configuración de temporizador variable: Se configura al timer 1 con un overflow de 1s, pero un contador de overflow regresivo configurable, permitiendo realizar una acción después de una cantidad especificada overflows consecutivos del temporizador.

III-B2. Manejo de estados: Utilizando este temporizador variable, se realiza la lógica de cambio de estado utilizando una máquina de estados y diferentes condicionales para ajustar por los distintos tipos de carga.

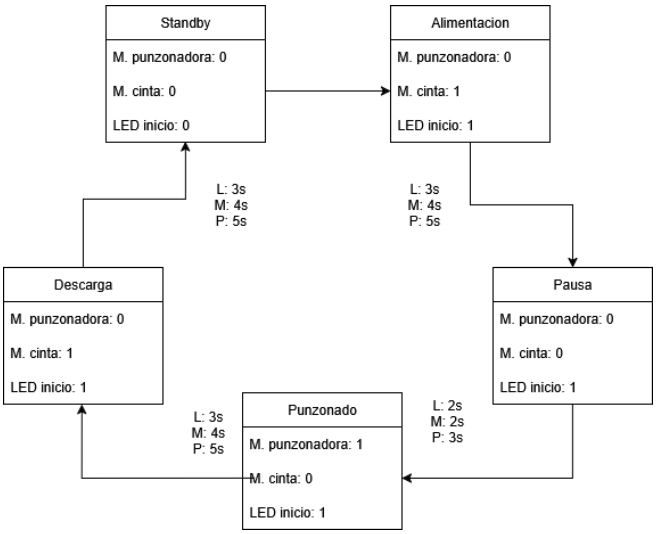

Figura 3. Diagrama de estados para punzonadora. Fuente: Elaboración propia.

III-B3. Botones, LEDs, y debouncing: Con la máquina de estados funcionando, se realiza un circuito indicador y de comando para poder iniciar el proceso, y configurar el tipo de carga que se va a manejar. Se implementa el timer 2 para realizar un deboucing de los pulsadores.

III-B4. USART: Implementado las funcionalidades de USART del anexo VII-P, se configura USART para mostrar un mensaje de inicio, y además indicar al usuario el estado actual de la máquina.

III-C. Matriz

III-C1. Objetivo inicial: El objetivo es controlar una **matriz de LEDs 8×8 (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-C2. 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.

III-C3. 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 e interrupciones. Los **frames** se guardan como mapas de bits 8×8 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.

III-C4. 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 100 ms. 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. Véase anexo VII-K

III-C5. 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. Véase anexo VII-S y anexo VII-P

III-D. Conversor

En este ejercicio se buscó visualizar en el osciloscopio la **señal 7**, correspondiente a una onda triangular (verificada a través de python en el anexo VII-F).

Para ello se empleó un conversor digital-analógico (DAC) basado en una red de resistencias R-2R, con un total de 17 resistencias de valores intercambiados entre 10k Ω y 20k Ω. El puerto D 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-E. Plotter

III-E1. Mapeo de comandos: Para el plotter se tomaron las instrucciones de la tabla mostrada en el anexo VII-E y se mapean las máscaras correspondientes a cada instrucción a variables internas para mejorar legibilidad de código.

III-E2. Interprete de secuencias: Utilizando las variables definidas previamente, se realizan series de instrucciones predefinidas para desplazamientos y figuras, guardando dichas secuencias en FLASH. Produciendo algo que podría ser descrito como un lenguaje de programación interno dentro del mismo programa, donde al plotter se le indicia el tipo de acción, y por cuando tiempo realizarla.

III-E3. Creación secuencia para círculo por Python: Para evitar realizar operaciones trigonométricas complejas con el microcontrolador debido a las limitaciones claras limitaciones del lenguaje, se implementa un programa en Python para utilizar operaciones trigonométricas e imprimir el "programa" equivalente en assembler para el plotter para obtener resultados con mejor resolución. Ver anexo VII-U

III-E4. USART: Una vez implementado el interprete de comandos, se configura la comunicación USART para mostrar un menú de opciones, y comandar al plotter a través de este para realizar diferentes trazados.

IV. Resultados

IV-A. Conversor digital-análogo

```
RESET:
    ; Apuntar Z al inicio de la LUT
    ldi ZH, HIGH(LUT_START<<1)
    ldi ZL, LOW(LUT_START<<1)

    ; Apuntar Y al final de la LUT
    ldi YH, HIGH(LUT_END<<1)
    ldi YL, LOW(LUT_END<<1)

MAIN_LOOP:
    ; Leer siguiente valor de la LUT
    ; y avanzar puntero
    lpm r16, Z+
    out PORTD, r16
    rcall delay

    cp  ZL, YL
    cpc ZH, YH
    ; Si no es el fin, seguir
    brne MAIN_LOOP

    ; Volver al inicio de la tabla
    ldi ZH, HIGH(LUT_START<<1)
    ldi ZL, LOW(LUT_START<<1)
    rjmp MAIN_LOOP
```

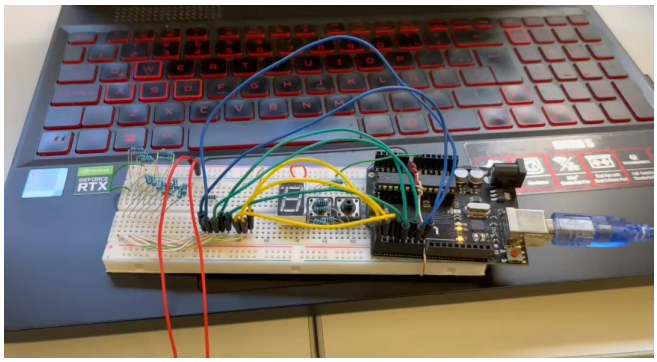

Figura 4. Circuito final para conversor digital-análogo. Fuente: [2].

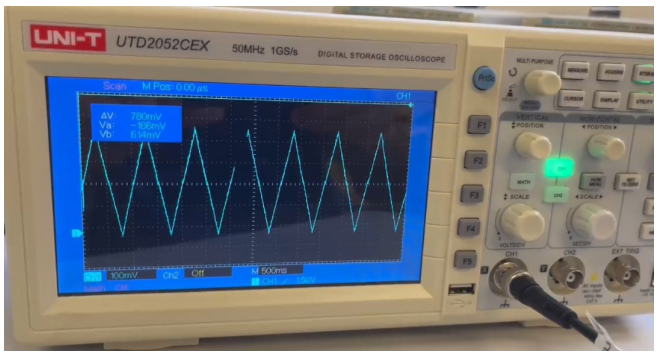

Figura 5. visualización de señal en osciloscopio. Fuente: [2].

IV-B. Matriz

IV-B1. Mapeado de puertos y pines: Se mapean los puertos y pines individuales del mismo modo al que se puede apreciar en el anexo VII-R.

IV-B2. Encendido de un solo LED:

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

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 adress

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
```

IV-B3. Multiplexado y dibujo de cuadros:

```
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
```

Manejo de cambio de estados utilizando USART para mostrar las diferentes imágenes en la matriz

```
USART_RX_ISR:
    lds r16, UDR0

    ; Apagar pantalla
    cpi r16, '0'
    breq USART_RX_ISR_CASE_0

    ; Texto desplazante
    cpi r16, '1'
    breq USART_RX_ISR_CASE_1

    ; ...

USART_RX_ISR_CASE_0:
    rcall SET_ANIMATION_START
    ; Disable timer interrupts
    ldi r16, 0b0 sts TIMSK2, r16
    ; Change state
    ldi r16, 0 mov current_state, r16
    rjmp USART_RX_ISR_END

USART_RX_ISR_CASE_1:
    rcall SET_ANIMATION_START
    ; Change state
    ldi r16, 1 mov current_state, r16
    ; Enable timer interrupts
    ldi r16, 0b1 sts TIMSK2, r16
    rjmp USART_RX_ISR_END

    ; ...
```

Aparte del manejo del cambio de estado, se implementó una máquina de estados, como la que se puede encontrar en el anexo VII-S.

```
MAIN:
    rcall STATE_MACHINE
    rjmp MAIN
```

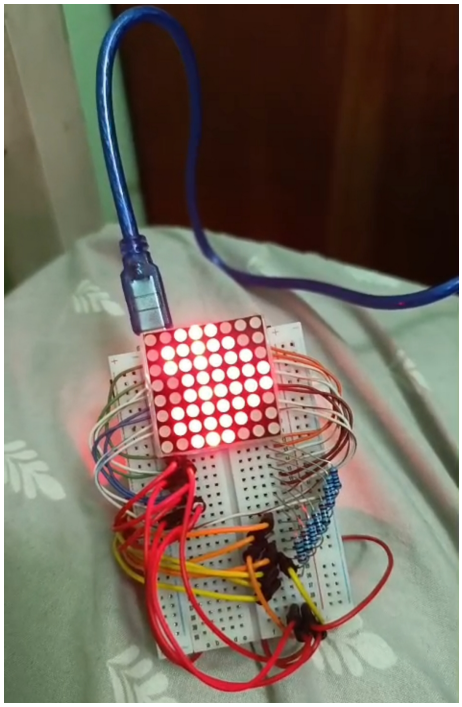

Figura 6. Circuito final para Matriz de LEDs. Fuente: [2].

V. Punzonadora

V-1. Configuración de temporizador variable:

```
.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
```

V-2. Manejo de estados: Control de estados general:

```
STATE_MACHINE:
cpi state, 0 breq STATE_MACHINE_STOP
cpi state, 1 breq STATE_MACHINE_ADVANCE
cpi state, 2 breq STATE_MACHINE_WAIT_1
cpi state, 3 breq STATE_MACHINE_PUNCH
cpi state, 4 breq STATE_MACHINE_WAIT_2
cpi state, 5 breq STATE_MACHINE_EXTRACT
rjmp STATE_MACHINE_END
```

Control de estados individual, cada caso tiene un manejo de tiempos específico dictado por el diagrama de estados mostrado con anterioridad.

```
STATE_MACHINE_STOP:
cpi load, 0 breq STOP_LOAD_0
cpi load, 1 breq STOP_LOAD_1
cpi load, 2 breq STOP_LOAD_2
rjmp STATE_MACHINE_STOP_SKIP
```

V-3. Botones, LEDs, y debouncing: Interrupción se descativa sola. Se vuelve a habilitar cuando la máquina de estados llega al final del recorrido:

```
INT0_ISR:
    push r16
    in r16, SREG
    push r16

    DISABLE_BUTTONS
    DISABLE_RX

    ldi state, 1
    ldi r16, 1
    sts event_pending, r16

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

Temporizador de deboucing para botones de cambio de carga. Timer 2 configurado para un overflow de aproximadamente 1 segundo (usando contador de overflow externo):

```
   T2_OVF_ISR:
push r16
   in r16, SREG
push r16

inc timer2_ovf_counter

ldi r16, _TIMER2_OVF_COUNT
   cp r16, timer2_ovf_counter
   brsh T2_OVF_ISR_END

DISABLE_TIMER_2
ENABLE_BUTTONS
clr timer2_ovf_counter

   T2_OVF_ISR_END:
 pop r16
 out SREG, r16
 pop r16
 reti
```

V-4. USART: Identico para los otros casos, se configura un menú que es enviado en el RESET del programa para ser mostrado al inicio, y una serie de condicionales determinan el comportamiento del sistema dependiendo del comando que recibe. En este caso: 1, 2, y 3 para seleccionar el tipo de carga. Y A para iniciar la secuencia del programa.

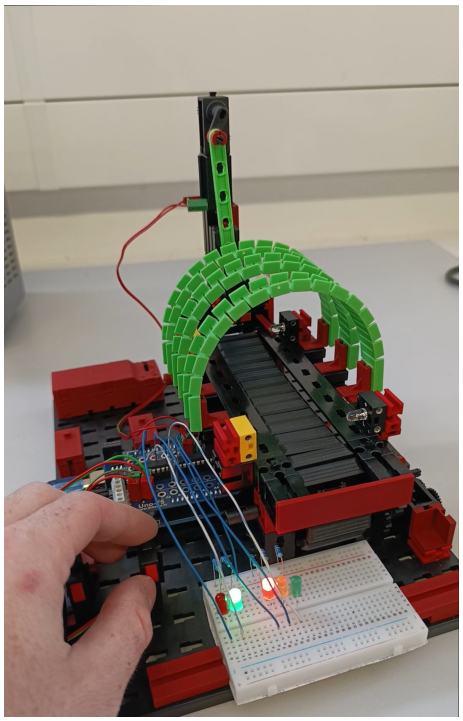

Figura 7. Ensamblado final de punzonadora. Fuente: [2].

V-A. Plotter

V-A1. Mapeo de comandos:

```
.equ SOLENOID_DOWN = 0b00000100
.equ SOLENOID_UP =  0b00001000
.equ DOWN =    0b00010000
.equ UP =    0b00100000
.equ RIGHT =   0b01000000
.equ LEFT =    0b10000000
.equ STOP =    0b00000000
```

V-A2. Interprete de secuencias:

```
DRAW:
    mov ZL, r16 mov ZH, r17
    DRAW_LOOP:
    lpm r18, Z+ ; Time
    lpm r19, Z+ ; Instruction

    out PORTD, r19 ; Send instruction

    cpi r19, STOP
    breq DRAW_END ; Stop drawing

    DRAW_TIMER_LOOP: ; Timer
    rcall S1
    dec r18 brne DRAW_TIMER_LOOP

    rjmp DRAW_LOOP ; Next instruction

    DRAW_END:
    ret
```

Nota: se encontró que para temporizadores con valores menores a 1ms no permitían el movimiento adecuado de los motores, por lo cual se decidió limitar la resolución con un temporizador de 1500 μS El siguiente ejemplo es de un programa hecho para el interprete creado para el plotter:

```
TRIANGLE_DATA:
    .db 5, SOLENOID_DOWN
    .db 20, SOLENOID_DOWN + RIGHT
    .db 10, SOLENOID_DOWN + UP + LEFT
    .db 10, SOLENOID_DOWN + DOWN + LEFT
    .db 5, SOLENOID_UP
    .db 5, STOP
```

V-A3. USART: Ejemplo de caso de interrupción de USART para dibujado de circulo:

```
; ...
USART_RX_ISR_CASE_5: ; CIRCLE
    ldi r16, low(CIRCLE_DATA<<1)
    ldi r17, high(CIRCLE_DATA<<1)
    rcall DRAW
    rjmp USART_RX_ISR_END
```

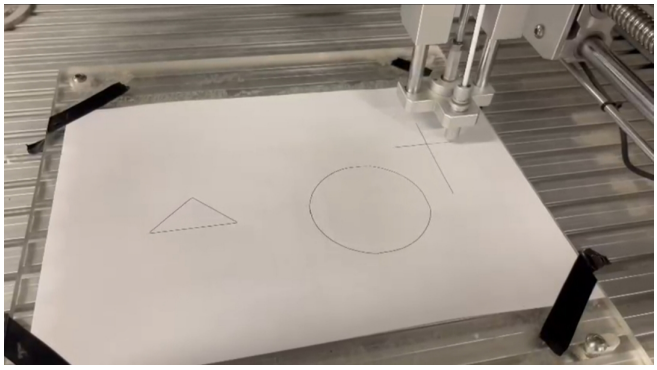

Figura 8. Figuras dibujadas en el plotter. Fuente: [2].

VI. Conclusiones

VI-A. Matriz

La matriz de *LEDs* presentó un desempeño sólido: permitió dibujar y desplazar patrones definidos en binario (unos y ceros) con buena estabilidad visual. La latencia de respuesta vía `USART` fue adecuada para la aplicación.

Mejoras propuestas: Incorporar modulación por ancho de pulso (PWM) por fila o por *pixel* para controlar el brillo de forma granular y habilitar imágenes/animaciones más complejas.

VI-B. Plotter

El plotter funcionó de manera satisfactoria. El intérprete permitió programarlo de forma intuitiva y se demostró que, mediante Python, es posible generar trayectorias avanzadas con curvas suaves. Se encontró la limitación de resolución de 1 mS como movimiento mínimo de los motores del plotter, al momento de buscar mejores trazados para el circulo.

Mejoras propuestas: Añadir PWM para un control más fino de la velocidad de los motores; permitir la selección del modo de temporización para ajustar la resolución según la figura; evaluar perfiles de aceleración para mejorar la calidad de las trayectorias.

VI-C. Conversor

El conversor mostró un desempeño correcto. La forma de onda obtenida fue clara y consistente con lo esperado según las gráficas del anexo VII-F.

VI-D. Punzonadora

La punzonadora operó con buen rendimiento. Se logró ajustar la carga y arrancar el proceso tanto por `USART` como mediante pulsadores físicos. El estado de la máquina y la carga seleccionada se reportaron en tiempo real por `USART`, con buena velocidad de respuesta.

Mejoras propuestas: Integrar sensores en la lógica de estados para que el sistema decida por eventos reales (objeto en cinta, fin de carrera) y no sólo por temporizadores; considerar enclavamientos de seguridad y manejo de fallas para aumentar la robustez.

Referencias

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

[2] Carpeta del laboratorio (google drive). Carpeta compartida del laboratorio. [Online]. Available: https://drive.google.com/drive/u/0/folders/1fP0aILozXeapRgDPDNWT1TRhAYr1PPPT

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

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

[5] Software Particles. (2024, Apr.) Learn how an 8×8 led display works and how to control it using an arduino. Figura: 8×8 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/

[6] Microchip Community (AVR Freaks). Avr freaks — comunidad de desarrolladores avr. Foros técnicos y soluciones prácticas sobre AVR. [Online]. Available: https://www.avrfreaks.net/

[7] J. Ganssle. A guide to debouncing. Referencia clásica para anti-rebote en pulsadores/encoders. [Online]. Available: https://www.ganssle.com/debouncing.htm

[8] G. Schmidt. (2021, Sep.) Beginners introduction to the assembly language of atmel-avr-microprocessors. Tutorial de avr-asm-tutorial.net (revisión septiembre 2021). [Online]. Available: https://kitsandparts.com/tutorials/assemblers/BeginnersAVRasm.pdf

[9] T. Redelberger. (2019, Apr.) Avr assembler for complex projects. Versión 0.4 (2019-04-06). [Online]. Available: https://web222.webclient5.de/doc/swdev/avrasm/advavrasm2/AdvancedUseAVRASM2_en_20190406.pdf

[10] Laboratorio de Microcontroladores, UTEC, "Repositorio de laboratorio de microcontroladores (tec.micro)," https://github.com/MateoLecuna/Tec.Micro, 2025, accedido el 26 de septiembre de 2025.

VII. Anexos

VII-A. Carpeta de laboratorio

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

VII-B. Repositorio del Laboratorio

Repositorio de Laboratorio de Microcontroladores (Tec.Micro) [10]

VII-C. Microcontrolador ATmega328P

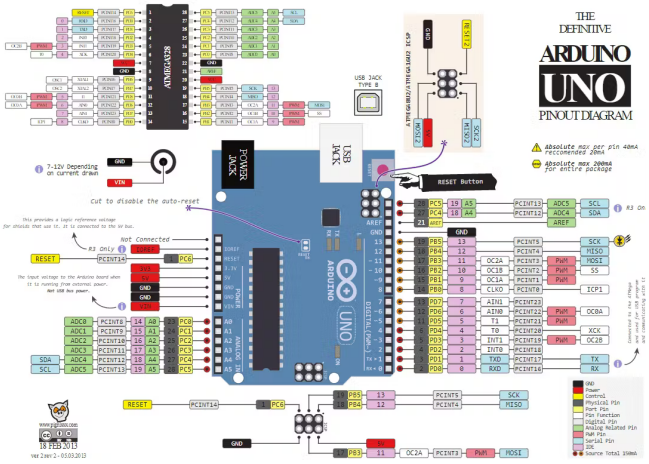

Figura 9. Diagrama de pines del Arduino Uno. Fuente: [4].

VII-D. Matriz de LEDs 1088AS

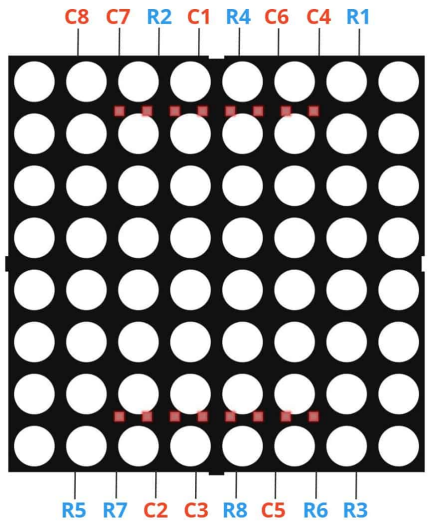

Figura 10. Pinout de Matriz de LEDs 1088AS. Fuente: [5].

VII-E. 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 11. Comandos del Plotter del laboratorio de Mecatrónica. Fuente: Hoja de laboratorio.

VII-F. Señal para conversor digital-analógico

```
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
```

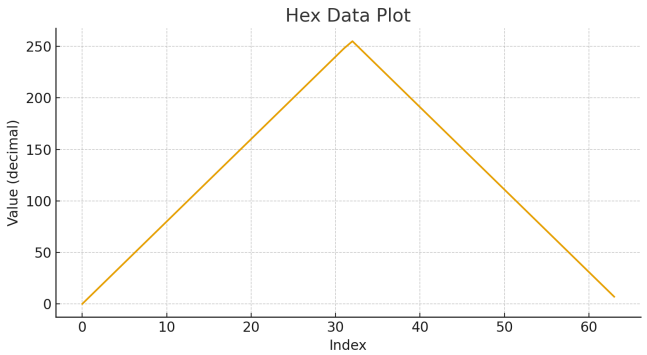

Figura 12. Graficado de señal para conversor. Fuente: Elaboración Própia

VII-G. 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

VII-H. Interrupt service routines

```
MI_ISR:
  push r16
```

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

VII-I. 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
```

VII-J. 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	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 13. Vectores de interrupciones en el ATmega328P. Fuente: hoja de datos del ATmega328P [3].

VII-K. Configuración 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 14. 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 15. 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 15)

Utilizando la tabla de vectores de interrupcion que se muestra en la figura 13 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
```

VII-L. Interrupciones Externas

VII-L1. 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 16. 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 17. 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 18. Registro de configucación de máscaras de interrupciones externas EICRA. Fuente: hoja de datos del ATmega328P [3].

VII-L2. 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 19. 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 20. Registro de configucación de mascara de pines para interrupciones PCIMSK0 (PCINT). Fuente: hoja de datos del ATmega328P [3].

VII-M. SRAM

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

VII-N. 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
```

VII-Ñ. USART Asíncrono

VII-Ñ1. 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
```

VII-Ñ2. Transmisión:

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

VII-Ñ3. 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
```

VII-O. 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.

VII-P. USART Asíncrono con Ring Buffer

VII-P1. 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
```

VII-P2. 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
```

VII-P3. 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
```

VII-P4. 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
```

VII-P5. Enviar mensajes por USART:

```
SEND_MESSAGE:
  push r16

  SEND_MESSAGE_LOOP:
  lpm r16, Z+
  cpi r16, 0

  breq SEND_MESSAGE_END
  rcall USART_WRITE_BYTE
  rjmp SEND_MESSAGE_LOOP

  SEND_MESSAGE_END:
  pop r16
  ret
```

Ejemplo de mensaje guardado en FLASH:

```
MSG_MENU:
.db "E. Actual: Standby", 0x0A, 0x0A
.db "Elija una opcion:", 0x0A
.db "[1] -> Conf. carga ligera ", 0x0A
.db "[2] -> Conf. carga mediana", 0x0A
.db "[3] -> Conf. carga pesada ", 0x0A
.db "[A] -> Iniciar", 0x0A, 0x0A, 0 ;Fin
```

VII-Q. Bit Masks

VII-Q1. 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
```

VII-Q2. 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
```

VII-R. 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
```

VII-S. 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
```

VII-T. 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"
```

VII-U. 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}")
```