



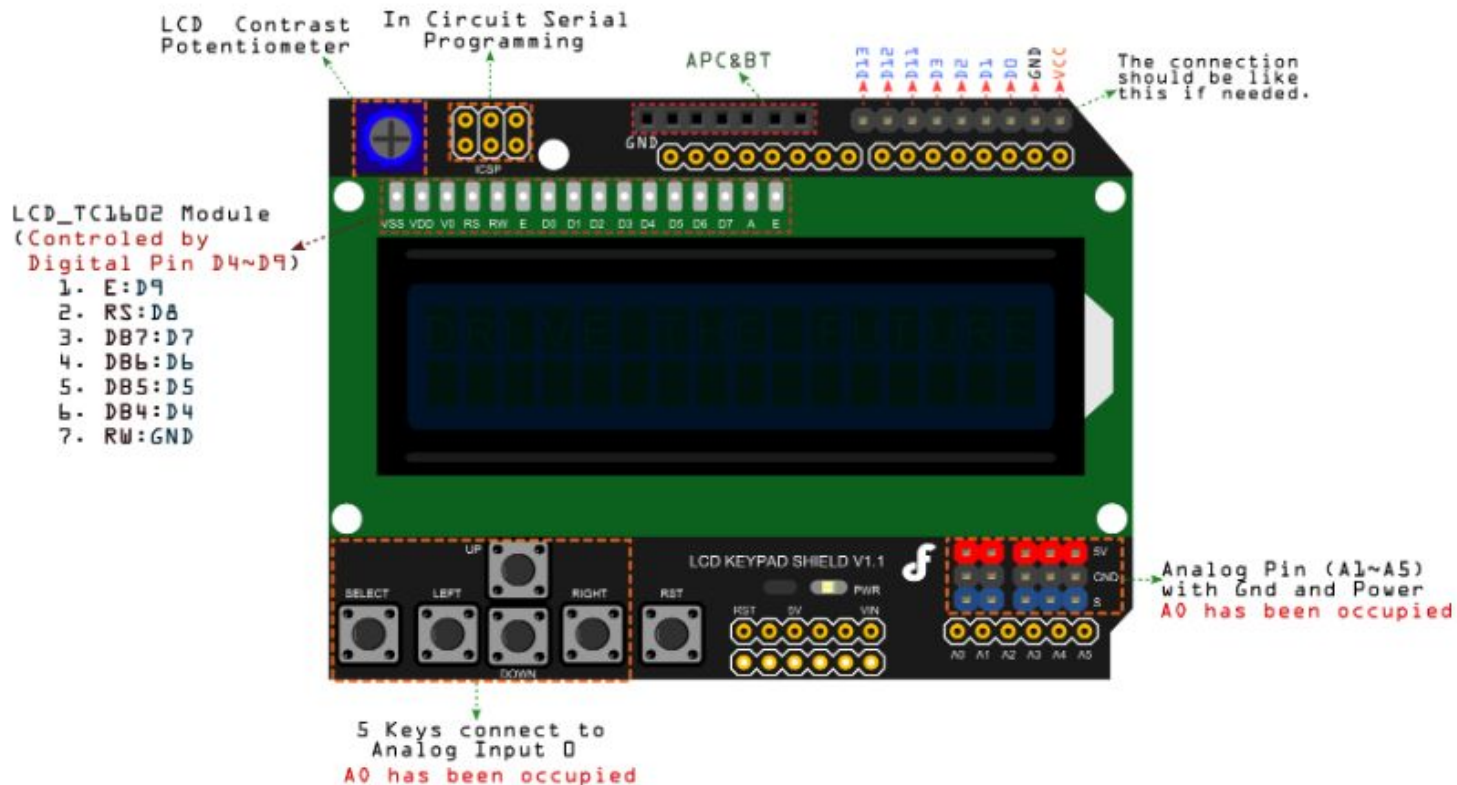
# Segundo Laboratorio

---

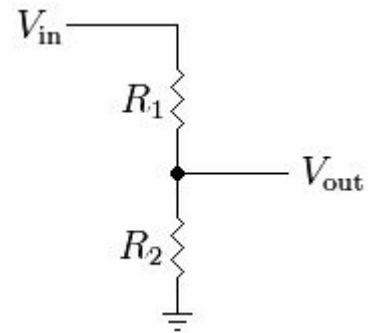
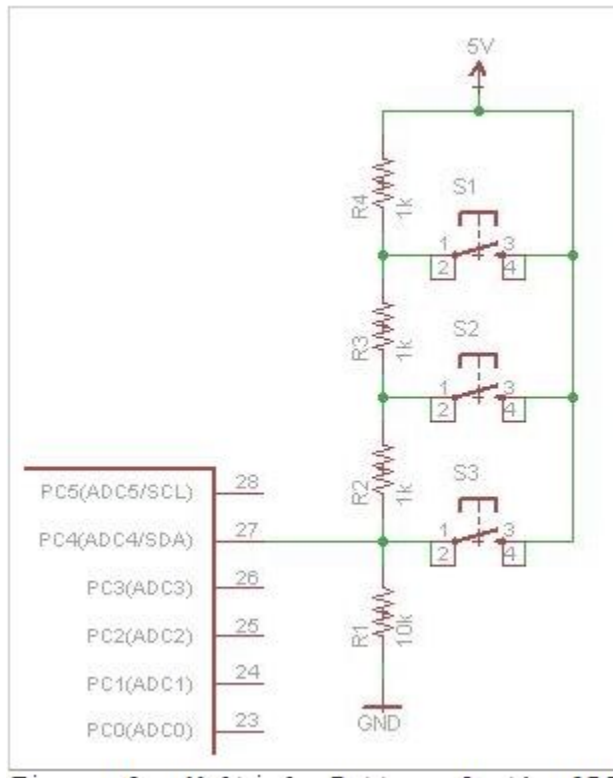
- LCD KeyPad Shield
- ADC
- Interrupciones
- Puntero a función
- Funciones de callback
- Cola de funciones
- Drivers

# LCDKeyPad Shield

## Board Overview



# LCDKeyPad Shield





# LCDKeyPad Shield

---

```
#include "Arduino.h"
// include the library code:
#include <LiquidCrystal.h>
// these constants won't change. But you can change the
size of
// your LCD using them:
const int numRows = 2;
const int numCols = 16;

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

//Key message
char msgs[5][17] = {" Right Key: OK ",
    " Up Key: OK ",
    " Down Key: OK ",
    " Left Key: OK ",
    " Select Key: OK "};

int adc_key_val[5] = {30, 150, 360, 535, 760 };
int NUM_KEYS = 5;
int adc_key_in;
int key=-1;
int oldkey=-1;
```

```
void setup() {
    pinMode(10, OUTPUT);
    // set up the LCD's number of columns and rows:
    lcd.begin(numCols,numRows);
    analogWrite(10, 100); //Controla intensidad backlight
    lcd.setCursor(0, 0);
    lcd.print("Key Pad Test");
    lcd.setCursor(0, 1);
    lcd.print("Sist.Emb. 2022");
    delay(2000);
    lcd.setCursor(0, 1);
    lcd.print("Press any key...");
}
```



# LCDKeyPad Shield

---

```
void loop()
{
  adc_key_in = analogRead(0); // read the value from the sensor

  key = get_key(adc_key_in); // convert into key press

  if (key != oldkey)          // if keypress is detected
  {
    delay(50);                // wait for debounce time
    adc_key_in = analogRead(0); // read the value from the sensor
    key = get_key(adc_key_in); // convert into key press
    if (key != oldkey)
    {
      oldkey = key;
      if (key >= 0)
      {
        lcd.setCursor(0, 1); //line=1, x=0
        lcd.print(msgs[key]);
      }
    }
  }
}
```

```
// Convert ADC value to key number
int get_key(unsigned int input)
{
  int k;
  for (k = 0; k < NUM_KEYS; k++)
    if (input < adc_key_val[k])
      return k;

  if (k >= NUM_KEYS)
    k = -1; // No valid key pressed

  return k;
}
```



# LCDKeyPad Shield

---

```
int analogRead(uint8_t pin)
{
    uint8_t low, high;
    if (pin >= 14) pin -= 14; // allow for channel or pin numbers
    ADMUX = (analog_reference << 6) | (pin & 0x07);
    // start the conversion
    sbi(ADCSRA, ADSC);

    // ADSC is cleared when the conversion finishes
    while (bit_is_set(ADCSRA, ADSC));

    // we have to read ADCL first; doing so locks both ADCL
    // and ADCH until ADCH is read.  reading ADCL second would
    // cause the results of each conversion to be discarded,
    // as ADCL and ADCH would be locked when it completed.
    low = ADCL;
    high = ADCH;

    // combine the two bytes
    return (high << 8) | low;
}
```

The analogRead() is defined in hardware/arduino/avr/cores/arduino/wiring\_analog.c



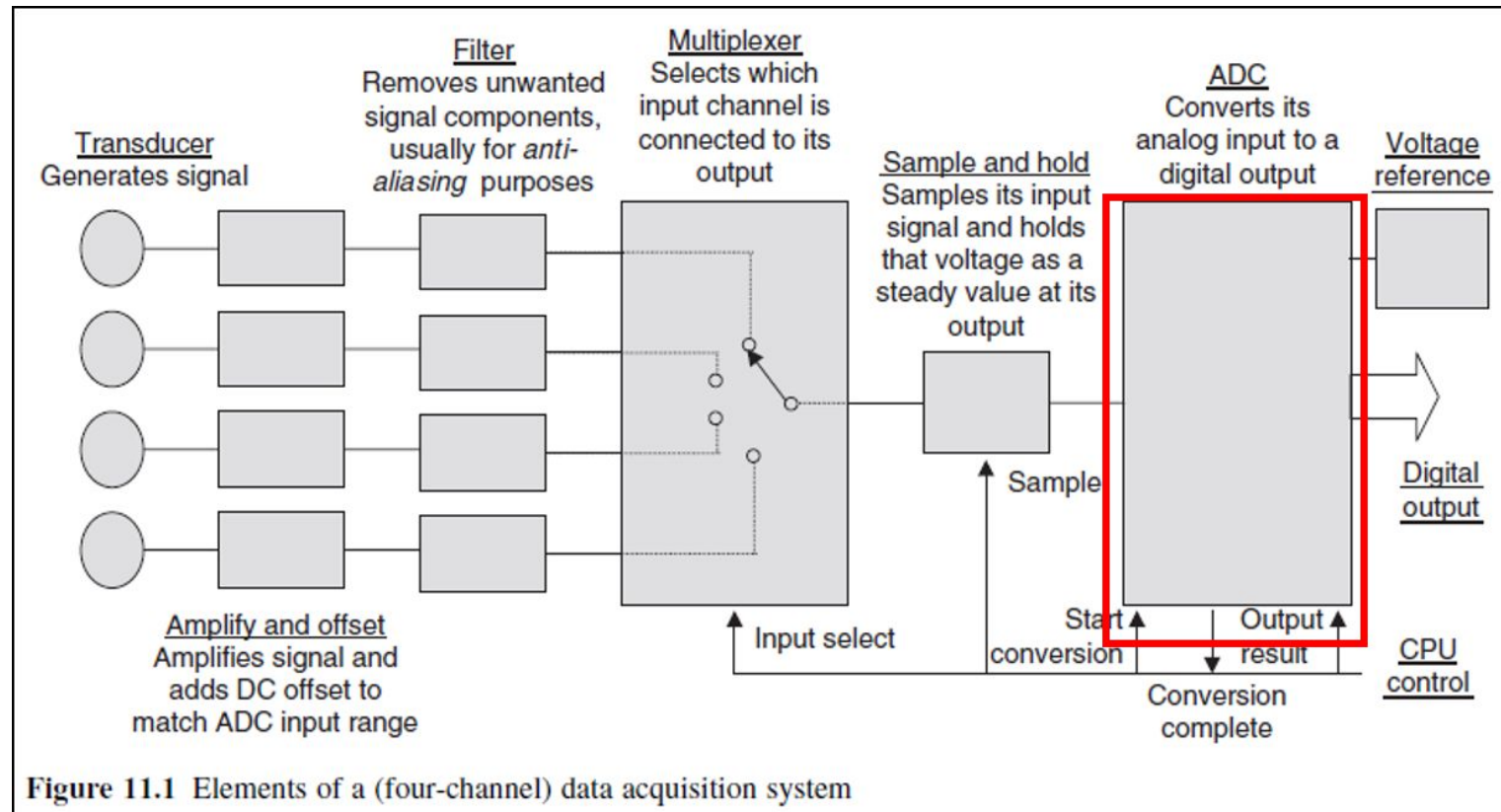
## ADC

---

Un ADC, o convertidor analógico a digital, permite **convertir un voltaje analógico en un valor digital** que puede ser utilizado por un microcontrolador. Hay sensores analógicos disponibles que miden la temperatura, la intensidad de la luz, la distancia, la posición y la fuerza, solo por nombrar algunos.

El ATmega328p presenta un ADC de aproximación sucesiva de 10 bits. **Tiene un ADC de 8 canales.** El ADC tiene un pin de voltaje de suministro analógico separado, **AVCC**. La referencia de voltaje puede estar desacoplada externamente en el pin AREF. AVCC se utiliza como referencia de voltaje. El ADC se puede configurar para que se **ejecute continuamente** (free running) o para que realice **solo una conversión**.

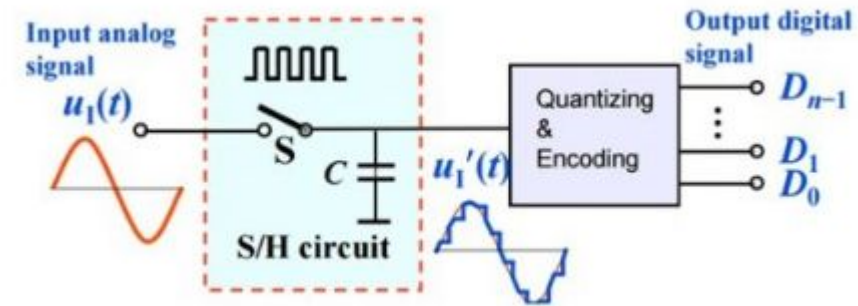
# ADC





# ADC

- S/H: Sampling and holding
- Q/E: Quantizing and Encoding

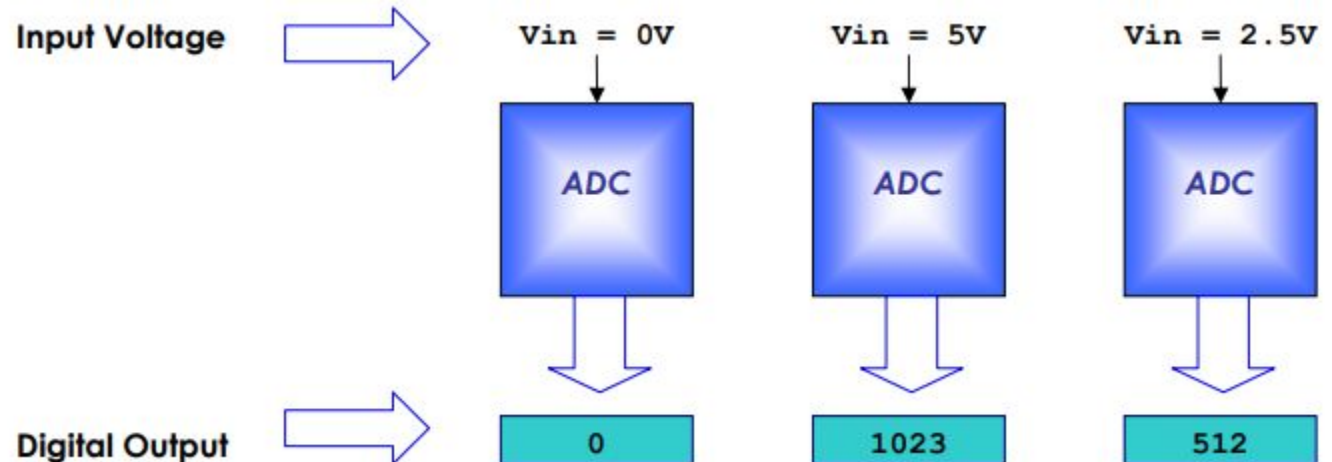


ADC process

# ADC

El **ADC** del ATMEGA328P tiene una **resolución de 10 bits**, (para esto cuando hablamos de los bits de resolución que tiene, cuanto más tenemos, mayor número de combinaciones podemos hacer) con lo que tendremos resultados con un rango de **0** (0v) como mínimo a **1023** (5v) como máximo. El ADC realiza la conversión por medio de aproximaciones sucesivas dentro del rango de referencia especificada  $V_{ref+}$  y  $V_{ref-}$ , donde  $V_{ref+}$  puede ser interna o externa y  $V_{ref-}$  comúnmente es 0. La resolución de cada bit está definida de acuerdo al rango de referencias de voltage, y queda dada por la fórmula:

$$\text{Resolución} = \frac{V_{ref+} - V_{ref-}}{2^n - 1}$$





## ADC

---

El valor calculado del ADC se obtiene con la siguiente fórmula

$$\text{Valor ADC} = \frac{2^n - 1}{V_{\text{ref+}}} \times V_{\text{in}}$$

Por lo cual podremos el valor del voltaje de entrada despejando  $V_{\text{in}}$

$$V_{\text{in}} = \text{Valor ADC} \times \frac{V_{\text{ref+}}}{2^n - 1}$$

$$V_{\text{in}} = \text{Valor ADC} \times \text{Resolución}$$



# ADC

---

El ADC necesita un clock de pulso para realizar la conversión. Requiere un frecuencia entre 50KHz y 200KHz para que la calidad de la conversión sea relativamente buena. Qué quiere decir esto ?

*A frecuencia altas la conversión es más rápida mientras que a frecuencia bajas la conversión es más exacta.*

1. 125kHz - 9.7bits or 10bits according to ATmega specifications
2. 250kHz - 9.5bits
3. 500kHz - 9.4bits
4. 1MHz - 8.75bits
5. 2MHz - 7.4bits
6. 4MHz - <6bits

Para generar la frecuencia de trabajo del ADC se tiene que programar el prescaler, que es un divisor de frecuencia que divide la frecuencia del microcontrolador, los posibles valores pueden ser 2, 4, 8, 16, 32, 64 o 128



## Registros del ADC

---

**ADMUX**: (ADC Multiplexer Selector Register) Es el registro en el que se selecciona la referencia de voltaje y el canal que se va a leer en el ADC.

**ADCSRA**: (ADC Control and Status Register A) En este registro de configuración del ADC, aquí se especifica el prescaler, la habilitación de interrupción, la habilitación del módulo y el inicio de conversión.

**ADC**: (ADC Data Register) Es un registro de 16 bits compuesto por 2 de 8 (ADCH y ADCL) y es donde se almacena el resultado de una conversión. Dependiendo del bit ADLAR en ADMUX, el ajuste de datos será hacia la izquierda o hacia la derecha:

# Cómo configurar el ADC

## ADC Multiplexer Selection Register – ADMUX

| Bit           | 7     | 6     | 5     | 4 | 3    | 2    | 1    | 0    |       |
|---------------|-------|-------|-------|---|------|------|------|------|-------|
|               | REFS1 | REFS0 | ADLAR | – | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write    | R/W   | R/W   | R/W   | R | R/W  | R/W  | R/W  | R/W  |       |
| Initial Value | 0     | 0     | 0     | 0 | 0    | 0    | 0    | 0    |       |

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in [Table 74](#). If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

**Table 74.** Voltage Reference Selections for ADC

| REFS1 | REFS0 | Voltage Reference Selection  |
|-------|-------|--|
| 0     | 0     | AREF, Internal $V_{ref}$ turned off                                  |
| 0     | 1     | $AV_{CC}$ with external capacitor at AREF pin                        |
| 1     | 0     | Reserved   |
| 1     | 1     | Internal 2.56V Voltage Reference with external capacitor at AREF pin |

# Cómo configurar el ADC

## Selección de multiplexor

The ADC Data  
Register – ADCL and  
ADCH

$ADLAR = 0$

| Bit           | 15   | 14   | 13   | 12   | 11   | 10   | 9    | 8    |      |
|---------------|------|------|------|------|------|------|------|------|------|
|               | –    | –    | –    | –    | –    | –    | ADC9 | ADC8 | ADCH |
|               | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADC1 | ADC0 | ADCL |
| Read/Write    | R    | R    | R    | R    | R    | R    | R    | R    |      |
| Initial Value | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |      |
|               | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |      |

$ADLAR = 1$

| Bit           | 15   | 14   | 13   | 12   | 11   | 10   | 9    | 8    |      |
|---------------|------|------|------|------|------|------|------|------|------|
|               | ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADCH |
|               | ADC1 | ADC0 | –    | –    | –    | –    | –    | –    | ADCL |
| Read/Write    | R    | R    | R    | R    | R    | R    | R    | R    |      |
| Initial Value | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |      |
|               | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |      |

### • Bits 3:0 – MUX3:0: Analog Channel Selection Bits

The value of these bits selects which analog inputs are connected to the ADC. See [Table 75](#) for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

**Table 75.** Input Channel Selections

| MUX3..0 | Single Ended Input |
|---------|--------------------|
| 0000    | ADC0               |
| 0001    | ADC1               |
| 0010    | ADC2               |
| 0011    | ADC3               |
| 0100    | ADC4               |
| 0101    | ADC5               |



## Cómo configurar el ADC

---

// Referencia y canal

**ADMUX = bit(REFS0) | adcPin; //01000000 | 00000001 = 01000001 = 0x41**

// Habilitar el ADC

**ADCSRA |= (1<<ADEN);**

// Prescaler a 128

**ADCSRA |= ((1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0));**

// Cuando usamos INTERRUPCIONES estos dos parámetros son importantes

**ADCSRA |= (1<<ADIE); // Set ADIE in ADCSRA to enable the ADC interrupt.**

**ADCSRA |= (1<<ADATE); // Set ADATE for free-running**

// Arrancar la conversión

**ADCSRA |= (1<<ADSC);**





## Cómo configurar el ADC

Cuando se completa una conversión de ADC, el resultado se encuentra en ADCH y ADCL.  
El registro de datos de ADC no se actualiza hasta que se lee ADCH.  
Primero debe leerse ADCL y luego ADCH.

| Bit           | 15   | 14   | 13   | 12   | 11   | 10   | 9    | 8    |      |
|---------------|------|------|------|------|------|------|------|------|------|
|               | —    | —    | —    | —    | —    | —    | ADC9 | ADC8 | ADCH |
|               | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADC1 | ADC0 | ADCL |
|               | 7    | 6    | 5    | 4    | 3    | 2    | 1    | 0    |      |
| Read/Write    | R    | R    | R    | R    | R    | R    | R    | R    |      |
|               | R    | R    | R    | R    | R    | R    | R    | R    |      |
| Initial Value | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |      |
|               | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |      |

*ADLAR = 0*

ADC Data Registers (ADLAR = 0)

Para leer los 10 bits del registro ADC se necesita hacer esto (depende del bit ADLAR) :

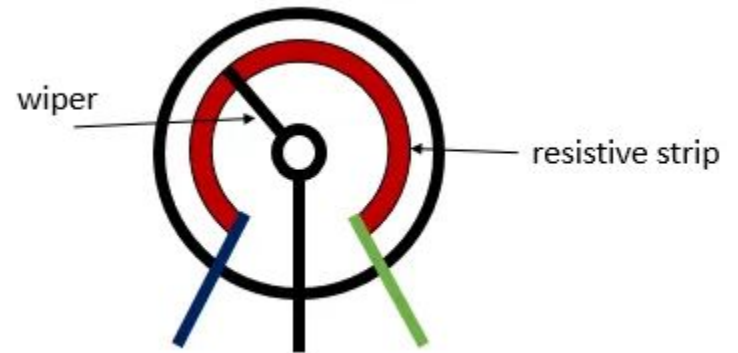
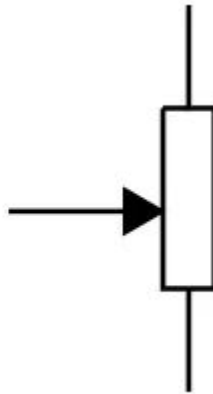
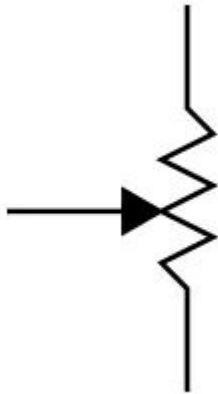
**$\text{analogVal} = \text{ADCL} | (\text{ADCH} \ll 8);$  (\* ver transparencia 6)**



## Ejemplo

---

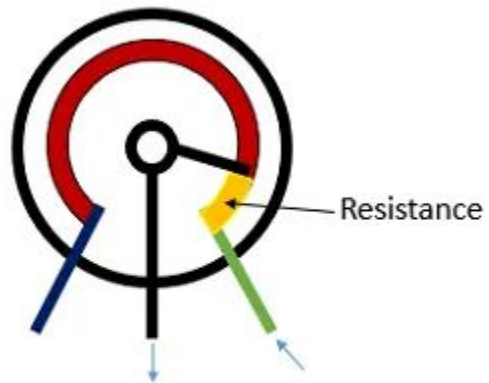
**Potenciometro:** puede ser usado como un dispositivo de entrada que permite ingresar un rango continuo de valores



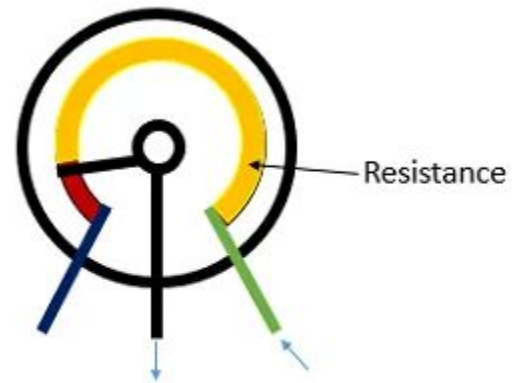


# Ejemplo

---



LESS RESISTANCE



MORE RESISTANCE

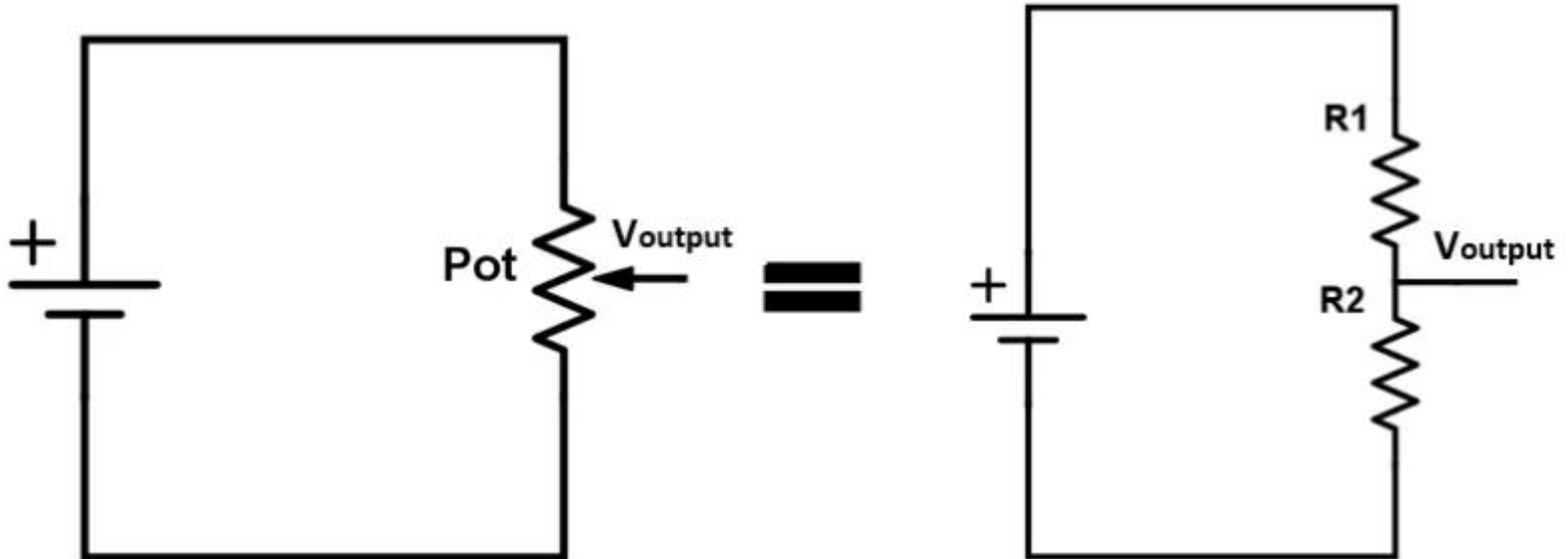


## Ejemplo

---

- Potenciómetros como divisores de voltaje
- Joystick ?

$$V_{out} = V_{cc} \times \frac{R_2}{R_1 + R_2}$$





## Ejemplo

---

Lee el valor analógico de un potenciómetro conectado a la entrada analógica 0 (ADC0) y se envía el resultado de la conversión por el puerto serie.

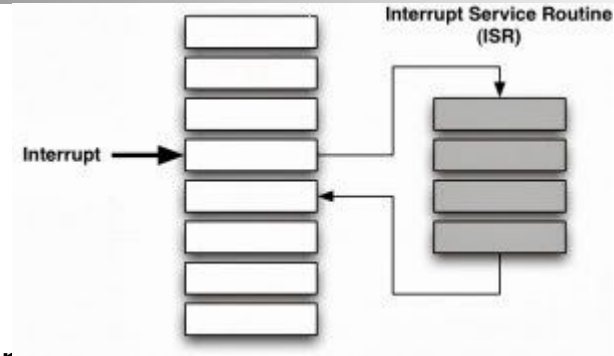
El micro está configurado a una frecuencia de 16 MHz y un prescaler de 128 para el ADC.

- ejemplo1

# Interrupciones

Un procesador normalmente tiene que

- medir tiempos
- generar eventos basados en tiempos
- responder en tiempo real a eventos que ocurren en tiempos impredecibles



Un sistema tiene que atender varias tareas y alguna de ellas periódicas y otras disparadas por eventos

Las **interrupciones** y los **timers** son herramientas **claves** para lograr un manejo efectivo del tiempo y los eventos en un **sistema embebido**



# Interrupciones

---

Las interrupciones son un recurso esencial de los sistemas embebidos. Básicamente, la interrupción es un mecanismo mediante el cual el CPU puede, ante cierto evento, **suspender lo que está haciendo en ese momento y pasar a atender una rutina de alta prioridad**. Una vez finalizada ésta, el CPU vuelve a su actividad anterior.

La interrupción es disparada por un evento externo al uC. Puede ser el cambio de estado de un pin (interrupción externa) o cierta señal de un dispositivo interno del uC, por ejemplo el desborde de un timer, el estado del ADC, etc



# Fuentes de interrupción

---

En el AVR, diversas condiciones pueden generar una interrupción, por ejemplo:

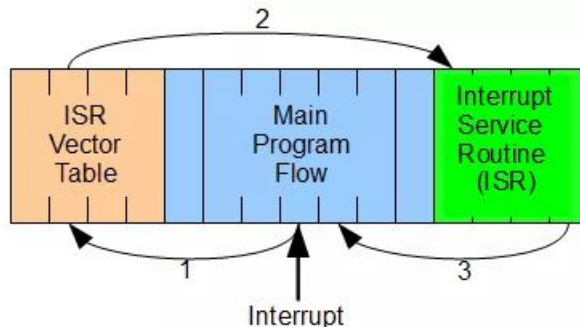
- El ADC, para indicar que está lista la conversión
- Los TIMERS, para indicar que se produjo un overflow u otra condiciones
- La UART, para indicar que llegó información por el bus.
- El cambio de estado de un pin de I/O - interrupción externa- para múltiples usos
- Poner un low en el pin de RESET



# Fuentes de interrupciones

Cada evento tiene asociada una dirección de memoria en donde se aloja el código específico a ejecutar, en la llamada tabla de vector de interrupción, que se ubica al comienzo de la memoria del programa.

En esa dirección el compilador coloca un jmp a la dirección de la ISR, que puede estar en cualquier otro lado.



| 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 OVFL  | 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 OVFL  | Timer/Counter1 overflow   |
| 15         | 0x001C          | TIMER0 COMPA | Timer/Counter0 compare match A  |
| 16         | 0x001E          | TIMER0 COMPB | Timer/Counter0 compare match B  |
| 17         | 0x0020          | TIMER0 OVFL  | Timer/Counter0 overflow   |



# Cómo definir un ISR

---

Para utilizar una ISR necesitamos hacer 3 cosas:

- 1) habilitar las interrupciones globales
- 2) habilitar la máscara de interrupción específica
- 3) **Escribir una función ISR y asociarla (linkearla) a esa interrupción**

Para definir una ISR, usamos estos nombres con combinación con la macro ISR, de la siguiente forma

```
#include <avr/interrupt.h>

ISR (INT0_vect) {

    //código a ejecutar

}
```

Ponemos el nombre tal como aparece en la tabla de vectores, bajo la columna "source" seguido de "vect"



## Ejemplo

---

Lee el valor analógico de un potenciómetro conectado a la entrada analógica 0 (ADC0), utilizando interrupciones y enviar el resultado de la conversión por el puerto serie.

- ejemplo2



# Puntero a función

---

- **Un puntero a función es una variable que almacena la dirección de una función.** Esta función se puede llamar más tarde, mediante el puntero. Este tipo de construcción es útil ya que encapsula el comportamiento, que se puede llamar mediante un puntero.
- La sintaxis para declarar un puntero a función es:  
**[return\_type] (\* pointer\_name) [(list\_of\_parameters)]**
- Ejemplo:
  - **int (\*ptr) (int,int);**
  - **void (\*function)();**



# Ejemplo

---

```
| punteroafuncion [REDACTED]
void (*punteroafuncion)();

uint8_t variable=10;

void setup()
{
    Serial.begin(9600);
    pinMode (13, OUTPUT);

    punteroafuncion = Blink05HZ;
}

void loop ()
{
    punteroafuncion();

    if (Serial.available())
    {
        cambiardefuncion();
    }
}

void Blink05HZ ()
```

- ejemplo3



# Puntero a función

---

- Objeto que apunta a una dirección de memoria
- Objeto que apunta a una dirección de una función en particular
- VENTAJAS:
  - Eficiencia ( no tener que hacer un IF en el loop )
  - Modulación (librerías, por ej..no depender un código de otro)



# Funciones de callback

---

- Una *función de callback* es una función que es pasada a otra función como un argumento
- Hay dos tipos de funciones de callback:
  - sincrónicas
  - asincrónicas
- Son comúnmente usados en sistemas manejados por eventos.
- En C, son implementadas mediante *punteros a funciones*



## Ejemplo

---

- Tenemos una librería del sensor (*driver*) que cuando algo ocurra nos avisa, llamando a una función de nuestro programa
- Pero como el programador que creó el sensor (driver) no sabe para que nosotros vamos a utilizar su librería..no sabe a qué función vamos a llamar.
  - Para eso utilizamos las *callbacks*.
  - ejemplo4

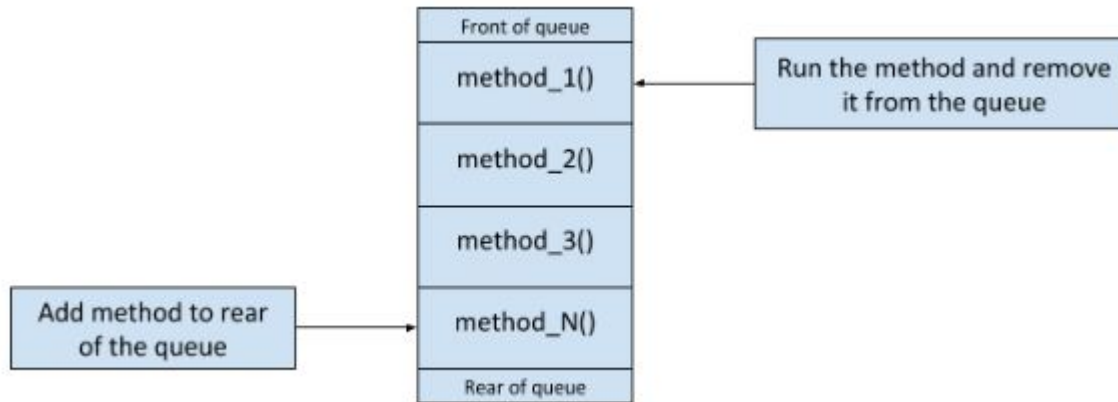




# Function Queue Scheduling

---

En la arquitectura de cola de funciones, las rutinas de interrupción agregan punteros de función a una cola de punteros de funciones.





## Function Queue Scheduling

---

En general las rutinas de interrupción agregan una función a la cola de funciones para reportar el evento.

Se evita llamar a funciones de callback desde rutinas de interrupción, porque se pierde el control de la duración y las actividades que se están realizando dentro de una rutina de interrupción.



## Function Queue Scheduling

---

- + Gran control sobre la prioridad
  - + Reduce la respuesta en el peor de los casos para el código de tarea de alta prioridad
  - + El tiempo de respuesta tiene una buena estabilidad en caso de cambios en el código.
- 
- Datos compartidos
  - Es posible que las tareas de baja prioridad nunca se ejecuten (starvation)



# Function Queue Scheduling

---

```
#include <stdbool.h>
#include <stdint.h>
#include "fnqueue.h"
#include "critical.h"
```

```
#define FNQUEUE_LENGTH 16
```

```
typedef void (*task)();
```

```
task fnqueue_functions[FNQUEUE_LENGTH];
```

```
bool fnqueue_init(void)
```

```
{
// Inicializa la cola
}
```

```
bool fnqueue_run(void)
```

```
{
// si hay una funcion la ejecuta
}
```

```
bool fnqueue_add(void (*function)(void))
```

```
{
// suma funciones a la cola
}
```

PLANIFICADOR



# Drivers

---

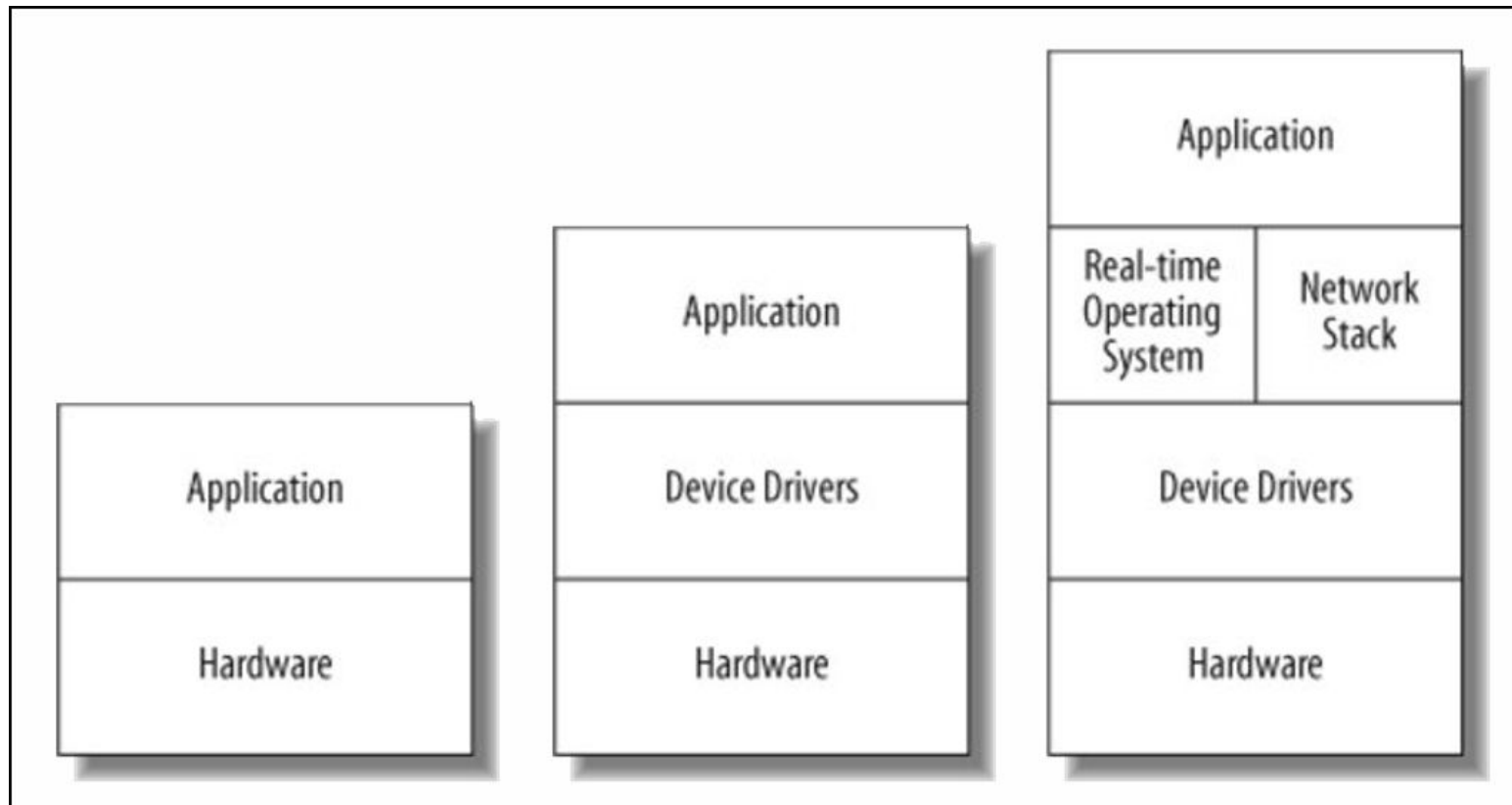
Son **porciones de software** destinadas a aislar al resto del sistema de los detalles específicos de un dispositivo particular de hardware.

El acceso al dispositivo se realiza exclusivamente mediante el **driver** (protección del recurso) y mediante una **interfaz** que abstrae de los detalles de implementación (capa de abstracción de hardware).

Se dan tanto en presencia de sistemas operativos (seguirán el modelo de drivers para el sistema particular) como en su ausencia (presentarán la forma de una API que el desarrollador podrá integrar en su sistema).



# Drivers





# Drivers

---

¿Cómo comenzar a construir un driver ?

- Identificar el hardware
- Identificar los eventos
- Identificar las ISR si tuviera



# Drivers

---

Cuando se trata de diseñar **DRIVERS**, siempre debe centrarse en un objetivo fácil de establecer: **ocultar el hardware completamente.**

Se desea que el driver del dispositivo sea la **única pieza de software** en todo el sistema que lea o escriba los registros de control y el estado del dispositivo en particular directamente.

Además, si el dispositivo genera cualquier interrupción, la rutina del servicio de interrupciones (**ISR**) que responde a ellas debe **ser una parte integral del dispositivo.**





# Drivers

---

## Beneficios

- Modularización.
- Como hay un sólo módulo que interactúa con el hardware, el estado del hardware se puede rastrear más fácilmente.
- Los cambios de software que resultan de los cambios del hardware se localizan solamente en el driver.



# Drivers

---

Usualmente consiste de 5 componentes:

- Inicializar una estructura de datos de control y estado de los registros del dispositivo.
- Un conjunto de variables para rastrear el estado actual del hardware.
- Una rutina para inicializar el hardware.
- Un conjunto de rutinas que proporcionan una API para los usuarios del driver del dispositivo.
- Una o más rutinas de servicio de interrupción.



# Ejemplo

---

device.h

```
#ifndef DEVICE_H
#define DEVICE_H
typedef struct
{ uint8_t param_a;
  bool (*callback_fn)(uint16_t datum);
} device_cfg;
```

device.cpp

```
/* Device driver template
#include "device.h"
static bool (* call_on_event)(uint16_t datum);
static volatile uint16_t datum;

bool device_init(device_cfg *configuration)
{
    call_on_event = configuration->callback_fn;
    // Device configuration
}
```

```
void send_signal(void)
{
    uint16_t datum_to_send;
    cli();
    datum_to_send = datum;
    sei();
    call_on_event(datum);
}

void device_isr()
{
    // Store datum from the device
    fnqueue_enqueue(send_signal);
}
```



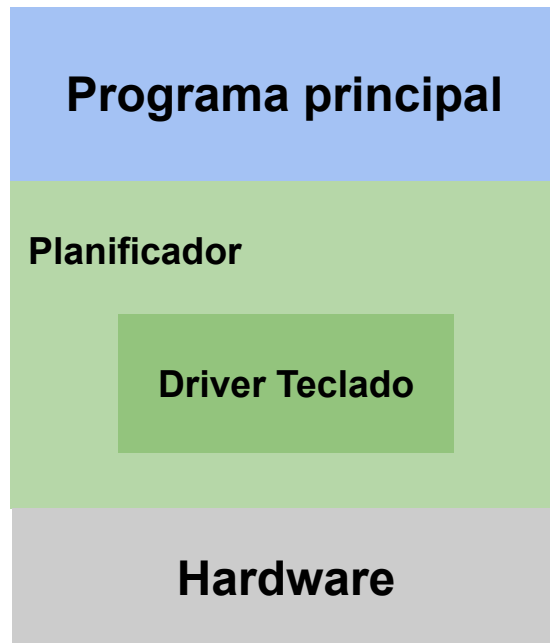
# Ejemplo

---

**Mala solución**



**Buena solución**





# Drivers

---

- Referencias
  - Programming Embedded Systems in C and C++ by Michael Barr (Chapt 7)