MICROCONTROLADORES Y ELECTRÓNICA DE POTENCIA

PROGRAMACIÓN DE MICROCONTROLADORES

Jordán, Emmanuel

Iriarte, Eduardo

Tabla de contenido

1. Motivación

Los sistemas mecatrónicos en general cuentan con mecanismos impulsados por *actuadores*, con el fin de cumplir con funciones de desplazamiento, posicionamiento, esfuerzo, etc. Estos actuadores pueden ser motores eléctricos, cilindros neumáticos o hidráulicos, etc., y se encargan de producir el movimiento relativo entre *eslabones* consecutivos del mecanismo, ubicándose ya sea directamente en las *articulaciones* (de rotación o traslación) que unen a los eslabones, o en la base del sistema y transmitiendo el movimiento a través de sistemas de transmisión (correas, poleas, engranajes, etc.).

Para lograr el comportamiento deseado del sistema es necesario realizar el *control* de los actuadores. Éste consta en general de un sistema electrónico que incluye una etapa de **procesamiento/control** y una etapa de **potencia**. La primera etapa en sus comienzos era analógica (ej. control PID mediante operacionales) y actualmente utiliza microcontroladores, procesadores digitales de señales (DSPs) y lógica programable. La segunda etapa depende del tipo y tamaño del actuador, y puede ser desde un simple transistor a puentes de transistores, los cuales normalmente trabajan en régimen de conmutación (como "llaves"), por motivos de eficiencia energética. Los fenómenos físicos que se producen en la etapa de potencia justifican muchos de los subsistemas presentes en los microcontroladores, particularmente los orientados al control de movimiento.

A continuación estudiaremos la programación de microcontroladores ***PIC***, ***AVR*** y con núcleo ***ARM*** analizando sus arquitecturas, características principales, módulos, registros, etc…, con ejemplos y consideraciones prácticas para poder no sólo simular su comportamiento sino también para implementarlos físicamente sin problemas.

Para la programación de los microcontroladores ***PIC*** utilizaremos el IDE "***PIC C Compiler***" y los grabaremos a través de un grabador de *PIC* (ejemplo PICkit 2 o 3). Experimentaremos sobre los *PIC* 12F675, 16F628A y 18F2550.

Figura 1. *Microcontroladores PIC12F675, PIC16F628A y PIC18F2550 de Microchip.*

Para la programación de los microcontroladores ***AVR*** utilizaremos el IDE "***Atmel Studio***" y los grabaremos a través de placas Arduino o de un programador de AVR (ejemplo USBISP). Experimentaremos sobre los AVR Attiny13A, Atmega328p y Atmega2560. Estos dos últimos los podemos usar montados sobre las placas *ArduinoUno* (o *ArduinoNano*) y *ArduinoMega*, respectivamente. No utilizaremos el IDE Arduino.

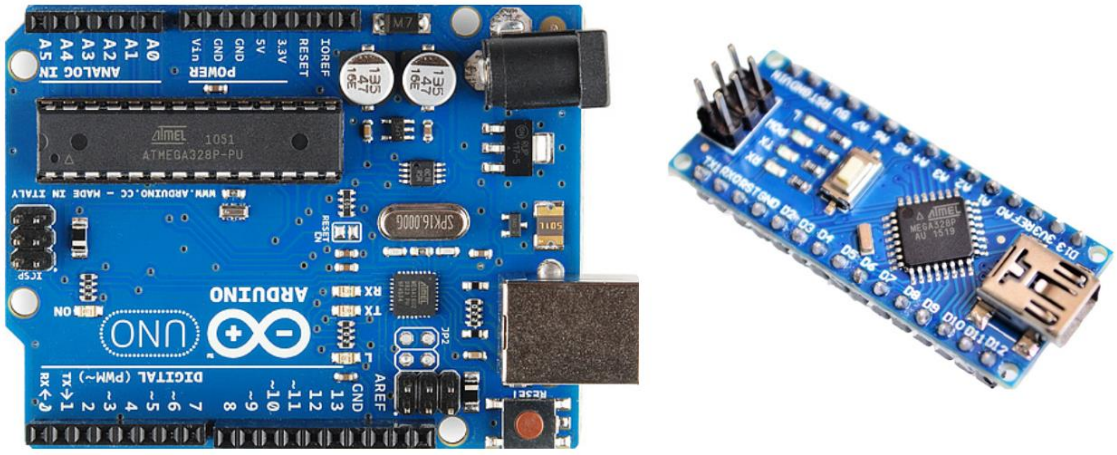


Figura 2. *Microcontroladores Atmega328p en placas ArduinoUno y ArduinoNano.*

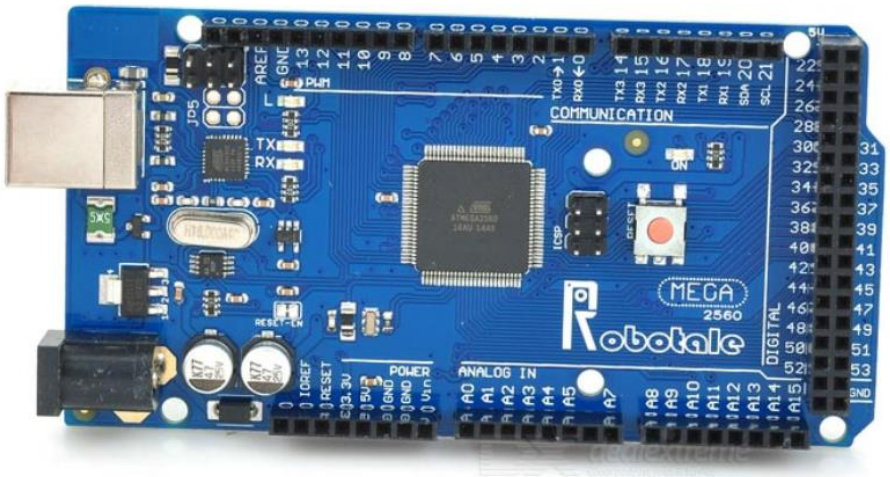


Figura 3. *Microcontrolador Atmega2560 en placa ArduinoMega.*

Se recomienda ver el apunte *"Instructivo AtmelStudio y AVR"* de la carpeta MyEP2020 [1].

En ese archivo además se muestran distintas formas de grabar/leer estos microcontroladores y de cambiar sus bits de configuración para adaptarlos a las necesidades de nuestro circuito.

Para la programación de los microcontroladores con núcleo ***ARM*** utilizaremos el IDE "***Atollic TrueSTUDIO***" y los grabaremos desde el mismo IDE con los mismos insertados en sus placas. Experimentaremos sobre el STM32F407VG (ARM Cortex-M4 de 32 bits) en la placa Discovery y sobre el STM32F103C8T6 (ARM Cortex-M3 de 32 bits) en la placa "BluePill".

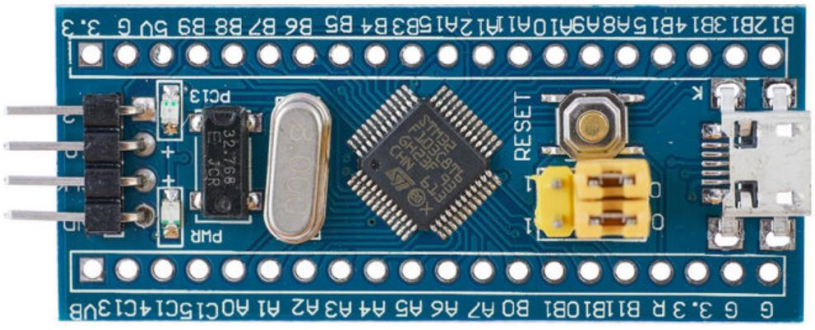


Figura 4. *Microcontrolador STM32F103C8T6 en placa "Blue Pill".*

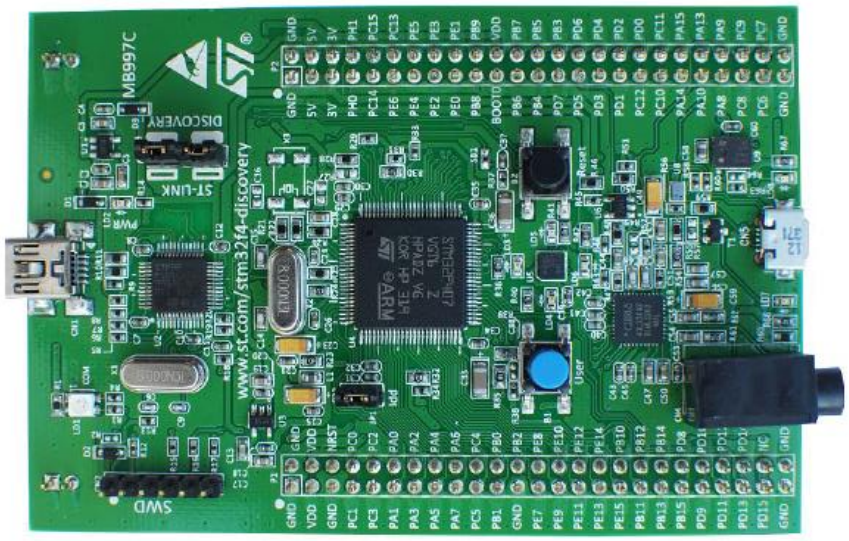


Figura 5. *Microcontrolador STM32F407 en placa Discovery.*

Si bien hay varias formas de grabar programas en los microcontroladores, experimentar en las placas comerciales que los traen montados es una forma práctica y didáctica de aprendizaje, debido al fácil acceso a los pines de los mismos y al aprovechamiento de elementos incorporados en las placas, como leds, pulsadores, cristal externo, etc. Por ejemplo, en las placas Arduino se aprovecha el cable USB que las vincula con la PC, para usarlo como medio de grabación, interfaz de comunicación serie y alimentación del circuito de baja potencia.

2. Introducción

Los microcontroladores ***PIC*** (Peripheral Interface Controller), fabricados por Microchip, tienen una arquitectura *RISC* (Reduced Instruction Set Computer) de 8 bits de tipo *Harvard* (separación física de datos e instrucciones). La memoria de *programa* está organizada en palabras de 12, 14 o 16 bits y la memoria de *datos* está compuesta por registros de 8 bits. Disponen de un set de instrucciones reducido (entre 33 y 77) de longitud fija (12, 14 o 16 bits). Admiten además instrucciones para acceder a cualquier bit de cualquier registro de la memoria de datos. Estos microcontroladores aplican la técnica de segmentado (pipeline) en la ejecución de las instrucciones en dos etapas, de modo que un ciclo de instrucción simple equivale a 4 ciclos del oscilador principal (las instrucciones de transferencia de control toman el doble de tiempo). La pila no forma parte de la memoria de datos, sino que ocupa un espacio independiente y tiene una profundidad limitada. Actualmente los *PIC* disponen de una gran variedad de periféricos incluidos (módulos de comunicación serie síncronos y asíncronos, temporizadores, conversores A/D y D/A, moduladores de ancho de pulso (PWM), interrupciones enmascarables, etc.) y de una memoria de programa de 512 a 32000 palabras.

Los microcontroladores ***AVR***, fabricados inicialmente por Atmel (adquiridos en la actualidad por Microchip), tienen una arquitectura *RISC* de 8 bits de tipo *Harvard*. El *programa* se ubica en la memoria FLASH y los *datos* están en 3 espacios diferentes: el archivo de registros (32 registros de 8 bits), en la SRAM y en la EEPROM. Las operaciones se realizan bajo un esquema registro-registro (carga-almacenamiento). El flujo del programa es secuencial, con incrementos automáticos del contador de programa (PC), y puede ser modificado con instrucciones de saltos condicionales o incondicionales y llamadas a rutinas, las cuales modifican al PC, permitiendo abarcar completamente el espacio de direcciones. La organización de la CPU permite solapar captura y ejecución de instrucciones, de modo que se ocupa un único ciclo del oscilador principal por cada instrucción simple.

Los microprocesadores con núlceo ***ARM*** (Advanced RISC Machine) tienen una arquitectura *RISC* de 32/64 bits de tipo *Harvard Modificada* (separación física de datos e instrucciones, pero con posibilidad de tratar instrucciones como datos). *ARM* no es una marca de un fabricante sino un núcleo licenciable (con cierta arquitectura, configuración de registros, etc.). Cada fabricante que compra estos núcleos (ejemplo STM) incorpora sus propios periféricos, con sus propios registros de configuración y uso, pero respetando las regiones de memoria. Tienen instrucciones de ejecución condicional y transferencia entre distintos registros en un solo ciclo, lo que permite obtener código muy eficiente (compacto y rápido). Tienen gran capacidad de direccionamiento (4GB en los de 32 bits). Los periféricos están mapeados en memoria (MMIO: Memory mapped I/O), con amplia franja de direcciones por periférico.

3. Manejo de registros en microcontroladores

Cada subsistema o periférico de los microcontroladores (TIMER, UART, A/D, D/A, GPIO, etc.) se puede activar/desactivar, verificar su estado, configurar para que funcione de cierto modo e intercambiar datos, etc. El manejo de los mismos se realiza mediante bits o conjuntos de bits que están en registros normalmente mapeados en memoria (es decir, estos registros se direccionan como una variable pero están en ubicaciones predeterminadas).

Los registros pueden ser de ***datos*** (r/w, ro, wo), de ***estado*** (ro, r/w) y de ***control*** (r/w). La división en ocasiones no es tan clara, especialmente en microcontroladores en los cuales la capacidad de direccionamiento es limitada. En general, el estado se verifica en bits individuales o "flags", mientras que el control (activar/desactivar/configurar) se realiza escribiendo bits individuales o conjuntos de bits.

Los pines de los microcontroladores permiten vincular sus sistemas internos con dispositivos externos. Existen bits de estado y de direccionamiento directamente asociados a los pines de propósito general, normalmente agrupados en ***puertos*** de 8 bits (también pueden ser 16 o 32), y se accede a ellos mediante registros mapeados en memoria. Normalmente los pines se configuran al inicio del programa como entrada o salida escribiendo en los bits de direccionamiento asociados.

3.1. Manejo de registros en PIC

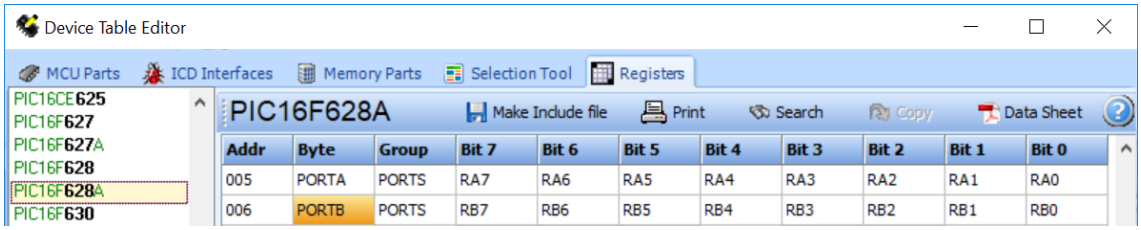
Como se comentó en la introducción, los *PIC* admiten instrucciones para acceder individualmente a cualquier bit de sus registros en la memoria de datos, sin modificar a sus bits vecinos. Por practicidad, se suelen definir etiquetas o rótulos tanto sobre los registros completos como sobre sus bits específicos, para luego utilizarlas en las distintas partes del programa y realizar determinadas acciones sobre ellos (encender, apagar, cambiar de estado, evaluar condición, etc.). Esto es opcional, pero suele mejorar la legibilidad de programa, como veremos en los próximos ejemplos.

Entre los registros especiales de los *PIC* se encuentran **PORTx** y **TRISx** (***x=A, B, C***,...), de 8 bits, de estado y de direccionamiento, respectivamente. Sus bits están directamente asociados a los pines físicos del *PIC*, agrupados en puertos de 8 pines, es decir, brindan una vinculación de software con hardware del microcontrolador. Al escribir un 1 o un 0 en sus bits, se están configurando a los pines asociados como:

TRISx → 1: entrada, 0: salida
PORTx → 1: encendido (Vdd), 0: apagado (GND)

Por ejemplo, al escribir un 0 en el bit 5 del TRISB (TRISB.5 = 0), y un 0 en el bit 5 del PORTB (PORTB.5 = 0), se está declarando como salida al pin 5 del puerto B, y asignándole un estado apagado.

Para hallar los nombres y las direcciones en memoria de los registros internos, en *PIC C Compiler*, se puede ir a "*View*" → "*Registers*". Por ejemplo, para el PIC16F628A, tenemos:



Como comentamos, será práctico definir etiquetas para usarlas en las distintas partes del programa. Esto se realiza mediante las directivas **#byte** o **#bit** y las direcciones en memoria correspondientes. Por ejemplo, si queremos llamar ***B0*** al bit 0 del registro PORTB y **dirB0** al bit 0 del registro TRISB del PIC16F628A, se escribe:

```
#byte PORTB = 0x006
#byte TRISB = 0x086
#bit  B0    = PORTB.0
#bit  dirB0 = TRISB.0
```

Así, se pueden usar los rótulos declarados de distintas formas:

```
dirB0 = 1;  //declara al pin B0 como entrada
dirB0 = 0;  //declara al pin B0 como salida
B0 = 0;     //apaga el pin B0
B0 = 1;     //enciende el pin B0
B0 != B0;   //cambia estado del pin B0
if(B0){...} //verifica estado del pin B0
```

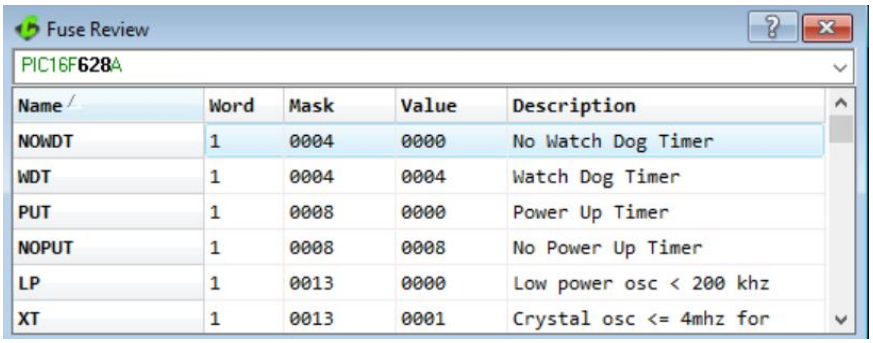
Ver consideraciones prácticas: "Pull-up, pull-down" [11.1] y "Dirección de pines por defecto" [11.3].

3.1.1. Ejemplo básico en PIC16F628A

Se debe controlar un led que se active mediante un pulsador A y se desactive mediante otro pulsador B. Primero se debe incluir la librería del *PIC* a utilizar, mediante:

```
#include <16F628A.h>
```

Luego se declaran los fusibles de configuración o "FUSE bits", opciones modificables al momento de grabar el micro. Para ver los fusibles disponibles para el *PIC* que se está utilizando, vamos a *"View"* → *"Config Bits"* y aparece la siguiente ventana:



Name	Word	Mask	Value	Description
NOWDT	1	0004	0000	No Watch Dog Timer
WDT	1	0004	0004	Watch Dog Timer
PUT	1	0008	0000	Power Up Timer
NOPUT	1	0008	0008	No Power Up Timer
LP	1	0013	0000	Low power osc < 200 khz
XT	1	0013	0001	Crystal osc <= 4mhz for

Elegimos trabajar con cristal interno y habilitar el pin MCLR, entre otras configuraciones que veremos posteriormente. Por lo tanto escribimos:

```
#FUSES INTRC          //Oscilador interno
#FUSES MCLR           //pin Master Clear habilitado
#FUSES NOWDT          //Sin timer de Watch Dog
#FUSES NOPUT          //Sin timer Power Up
#FUSES NOPROTECT      //Código no protegido contra lectura
#FUSES NOBROWNOUT     //Sin reseteo Brownout
#FUSES NOLVP          //Sin programación con bajo voltaje B3(PIC16) o B5(PIC18)
#FUSES NOCPD          //Sin protección EE
#FUSES RESERVED       //Usado para setear los FUSE bits reservados
```

Nota: al habilitar MCLR, el pin 4 de este *PIC* debe estar en estado alto para que no se resetee (se lo suele conectar con una resistencia de pull-up de $10k\Omega$ y un capacitor de $0.1\mu F$ a GND como filtro pasa bajo, ver esquemático en ejemplos).

Especificamos la frecuencia de trabajo del micro (en este caso, cristal interno de 4MHz):

```
#use delay(internal=4000000)
```

Nota: en los *PIC* un ciclo de instrucción simple (tiempo entre ejecución de instrucciones consecutivas) equivale a 4 ciclos del oscilador principal (cristal interno o externo), mientras que un ciclo de instrucción compleja (instrucciones de saltos, etc.) equivale a 2 ciclos de instrucción simple. Es decir, la frecuencia de trabajo es un cuarto de la frecuencia del oscilador principal. En este ejemplo:

$$F_{OSC} = 4MHz \rightarrow F_{CY} = F_{OSC}/4 = 1MHz \rightarrow T_{CY} = 1/F_{CY} = 1\mu s$$

Luego elegimos tres pines de propósito general, dos como entradas para los pulsadores A y B (elegimos PORTB6 y PORTB7, respectivamente) y uno como salida para el led (elegimos PORTA0). Para esto será útil definir los siguientes rótulos:

```
#byte PORTA = 0x005
#byte TRISA = 0x085
#byte PORTB = 0x006
#byte TRISB = 0x086
#bit  A0    = PORTA.0
#bit  B6    = PORTB.6
#bit  B7    = PORTB.7
#bit  dirA0 = TRISA.0
#bit  dirB6 = TRISB.6
#bit  dirB7 = TRISB.7
```

El programa normalmente tendrá una parte de inicialización en la entrada al main (en este caso indicando la frecuencia del oscilador, y declarando el estado y direccionamiento de los pines) y un ciclo infinito (en este caso testeando el estado de los pulsadores).

```
void main()
{
   setup_oscillator(OSC_4MHZ);
   dirA0 = 0;    // el bit 0 del puerto A es SALIDA
   A0    = 0;
   dirB6 = 1;    // el bit 6 del puerto B es ENTRADA
   dirB7 = 1;    // el bit 7 del puerto B es ENTRADA

   while(TRUE)  // ciclo infinito
   {
      if(B6)      A0 = 1;  // pone en estado ALTO a PORTA0. equivalente: output_high(pin_A0);
      else if(B7) A0 = 0;  // pone en estado BAJO a PORTA0. equivalente: output_low(pin_A0);
   }
}
```

En este ejemplo, el orden de verificación de las entradas en el ciclo infinito da prioridad al pulsador A sobre el pulsador B. De manera equivalente, se podría haber realizado la inicialización de pines operando sobre los registros completos de estado y direccionamiento, como sigue:

```
TRISA = 0b00000000; //Todos los pines del puerto A son salidas
PORTA = 0b00000000; //Todos los pines del puerto A apagados
TRISB = 0b11000000; //Todos los pines del puerto B son salidas, excepto B6 y B7
PORTB = 0b00000000; //Todos los pines del puerto B apagados
```

Nota: Si queremos analizar el programa creado por el compilador (lenguaje ensamblador), el cual se grabará en el *PIC*, luego de compilar vamos a *"View"* → *"C/ASM List"*.

3.2. Manejo de registros en AVR

En los microcontroladores *AVR* no se puede acceder individualmente a los bits de sus registros de la misma manera que en los *PIC*, sino únicamente al registro completo. Por lo tanto, para modificar un bit sin modificar a sus vecinos se usan operaciones de enmascaramiento. Para más detalles, ver apunte "*Operadores de manejo de bits en C*" de la carpeta MyEP2020 [1].

En estos micros no es necesario definir etiquetas de sus registros o bits a través de sus direcciones en memoria, como en los *PIC*, sino que directamente se escriben sus nombres como aparecen en el catálogo del microcontrolador (ya que están definidos en la librería del dispositivo correspondiente). Pero sí será útil definir etiquetas para cada una de las situaciones en las que se usen los bits.

Entre los registros especiales de los *AVR* se encuentran **PORTx** y **DDRx** (***x*=*A, B, C***,...), de 8 bits, de estado y de direccionamiento, respectivamente. Sus bits están directamente asociados a los pines físicos del micro, agrupados en puertos de 8 pines, es decir, brindan una vinculación de software con hardware del microcontrolador. Al escribir un 1 o un 0 en sus bits, se están configurando a los pines asociados como:

DDRx → 0: entrada, 1: salida (contrario a TRISx en los *PIC*)
PORTx → 1: encendido (Vdd), 0: apagado (GND)

Por ejemplo, al escribir un 1 en el bit 5 del DDRB, y un 0 en el bit 5 del PORTB, se está declarando como salida al pin 5 del puerto B, y asignándole un estado bajo de tensión.

El registro de verificación **PINx** (***x*=*A, B, C***,...) permite leer el estado de los pines (se actualiza un ciclo de reloj después de escribirse el registro PORTx correspondiente).

PINx → 1: encendido (Vdd), 0: apagado (GND) (lectura)

Nota: no se usa el mismo registro para leer y escribir (como en los *PIC* con sus registros PORTx). En los *AVR* se usa PINx para lectura y PORTx para escritura.

Por ejemplo, para operar sobre el bit 5 del puerto B del Atmega328p, será práctico definir etiquetas como las siguientes:

```
#define B5_ON     (PORTB |=  (1<<PORTB5))   // Enciende B5
#define B5_OFF    (PORTB &=~ (1<<PORTB5))   // Apaga B5
#define B5_OUT    (DDRB  |=  (1<<DDB5))     // B5 es Salida
#define B5_IN     (DDRB  &=~ (1<<DDB5))     // B5 es Entrada
#define B5_Toggle (PORTB ^=  (1<<PORTB5))   // Cambia estado de B5
#define B5_TEST   (PINB   &  (1<<PINB5))    // Verifica estado de B5
```

Notar que no fue necesario definir etiquetas para los registros ni para sus bits, sino que simplemente se accedió a ellos por sus nombres. Con estas definiciones, se puede por ejemplo declarar como salida PB5 y darle un estado inicial alto, escribiendo en el programa:

```
B5_OUT;
B5_ON;
```

Luego se puede evaluar su estado escribiendo:

```
if(B5_TEST){...}
```

Ver consideraciones prácticas: "Pull-up, pull-down" [11.1] y "Dirección de pines por defecto" [11.3].

3.2.1. Ejemplo básico en Atmega328p

Se debe controlar un led que se active mediante un pulsador A y se desactive mediante otro pulsador B.

En primer lugar, se debe tener en cuenta que en los microcontroladores *AVR* los FUSE bits no se pueden modificar desde el código fuente (como en los *PIC*). Para más detalles, ver apunte "*Instructivo AtmelStudio y AVR*" de la carpeta MyEP2020 [1].

Luego se debe especificar la frecuencia de trabajo del micro. Por ejemplo, las placas ArduinoUno/Nano (con Atmega328p) y ArduinoMega (con Atmega2560) por lo general traen incorporado un cristal externo de 16MHz. En esos casos definimos:

```
#define F_CPU 16000000
```

Nota: en los *AVR* un ciclo de instrucción simple (tiempo entre ejecución de instrucciones consecutivas) equivale a un ciclo del oscilador principal (cristal interno o externo), mientras que un ciclo de instrucción compleja (instrucciones de saltos, etc.) equivale a 2 ciclos de instrucción simple. Es decir, la frecuencia de trabajo es igual a la frecuencia del oscilador principal. En este ejemplo:

$$F_{OSC} = F_{CY} = 16MHz \rightarrow T_{CY} = 1/F_{CY} = 62.5ns$$

A continuación se incluyen las librerías necesarias para el funcionamiento del programa. En este caso necesitamos únicamente "*avr/io.h*", la cual contiene las definiciones de puertos y registros para cada micro.

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

Luego elegimos tres pines de propósito general, dos como entradas para los pulsadores A y B (elegimos PORTB3 y PORTB4, respectivamente) y uno como salida para el led (elegimos PORTB5). Para esto será útil definir los siguientes rótulos:

```
#define PulsadorA_Entrada   (DDRB  &=~ (1<<DDB3))
#define PulsadorB_Entrada   (DDRB  &=~ (1<<DDB4))
#define PulsadorA_Test      (PINB   &  (1<<PINB3))
#define PulsadorB_Test      (PINB   &  (1<<PINB4))
#define Led_ON              (PORTB |=  (1<<PORTB5))
#define Led_OFF             (PORTB &=~ (1<<PORTB5))
#define Led_Salida          (DDRB  |=  (1<<DDB5))
```

El programa normalmente tendrá una parte de inicialización en la entrada al main (en este caso declarando el estado y direccionamiento de los pines) y un ciclo infinito (en este caso testeando el estado de los pulsadores para actuar sobre el led).

```
int main(void)
{
    PulsadorA_Entrada;      //PORTB3 (pin 11 de ArduinoUno) es entrada
    PulsadorB_Entrada;      //PORTB4 (pin 12 de ArduinoUno) es entrada
    Led_OFF;                //PORTB5 (pin 13 de ArduinoUno, led de placa) apagado
    Led_Salida;             //B5 es salida
    while (1)   // ciclo infinito
    {
        if(PulsadorA_Test)       Led_ON;  //Si B3 tiene un estado alto, enciende B5
        else if (PulsadorB_Test) Led_OFF; //Sino, y si B4 tiene un estado alto, apaga B5
    }
}
```

En este ejemplo, el orden de verificación de las entradas en el ciclo infinito da prioridad al pulsador A sobre el pulsador B. De manera equivalente, se podría haber realizado la inicialización de pines operando sobre los registros completos de estado y direccionamiento, como sigue:

```
DDRB  = 0b11100111; //Todos los bits del puerto B SALIDAS, excepto B3 y B4
PORTB = 0b00000000; //Todos los bits del puerto B APAGADOS
//Todos los bits restantes, APAGADOS y como SALIDAS
DDRC = 0xFF; PORTC = 0x00;
DDRD = 0xFF; PORTD = 0x00;
```

3.3. Manejo de registros en STM32

En estos microcontroladores, los pines de propósito general (*GPIO*) tienen más parámetros de configuración respecto de los estudiados en las subsecciones anteriores. Cada puerto de entrada/salida ***GPIOx*** (***x=A,B,C,***....) contiene grupos de 16 pines y tiene asociados registros de 32 bits de:

- ***Configuración***:

GPIOx_MODER → MODO: entrada digital (00), salida (01), alternado (10) o entrada analógica (11)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIOx_OTYPER → TIPO SALIDA: push-pull (0) o colector abierto (1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIOx_OSPEEDR → VELOCIDAD SALIDA: baja (00), media (01), alta (10) o muy alta (11). Cambia la capacidad de drenar corriente. Se elige baja velocidad cuando se busca que la conmutación del pin no introduzca ruido en la línea (interferencia).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIOx_PUPDR → PULL-UP/DOWN: ninguna (00), pull-up (01) o pull-down (10)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- ***Datos***:

GPIOx_IDR → DATOS ENTRADA (bits sólo lectura, con acceso en modo palabra)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

GPIOx_ODR → DATOS SALIDA (bits escritura/lectura, con acceso en modo palabra)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- ***Set/Reset***:

GPIOx_BSRR → SET/RESET atómico de bits

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Los primeros 16 bits corresponden al encendido y los 16 restantes al apagado del pin asociado al bit del puerto. Ambas acciones se logran escribiendo un 1 en el lugar correspondiente.

- ***Bloqueo***:

GPIOx_LCKR

- ***Selección de funciones alternadas***:

GPIOx_AFRR
GPIOx_AFRL

Además, para utilizar los pines de *GPIO* se debe inicialmente habilitar al bus *AHB1* (Advanced High performance Bus 1) el cual puede acceder a los periféricos, en particular al *GPIO*. Para esto buscamos el registro *AHB1ER* (AHB1 Enable Register), el cual se encuentra en el grupo de registros *RCC* (Reset Clock & Control).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	**DMA2DEN**	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

3.3.1. Ejemplo básico en STM32F407

En el siguiente ejemplo se enciende y apaga el led verde de la placa Discovery (PD12), según el estado de un pulsador en el pin PC7, configurado con resistencia de pull-up interna. Se muestra a continuación cómo se puede lograr esto escribiendo los registros involucrados, descriptos anteriormente.

```
#include "stm32f4xx.h"
int main(void)
{
    //Habilitación del clock de GPIO para los puertos D y C por bus AHB1
    RCC->AHB1ENR   |=  (3<<2);
    //Configuración del pin 12 del puerto D (salida a escribir)
    int pinSalida = 12;
    GPIOD->MODER   &=~ (1<<(pinSalida*2+1));
    GPIOD->MODER   |=  (1<<pinSalida*2); //Modo SALIDA     (01 en bits MODER6[1:0])
    GPIOD->OTYPER  &=~ (1<<pinSalida);   //Modo Push-pull  ( 0 en bit  OT6)
    GPIOD->OSPEEDR &=~ (3<<pinSalida*2); //Velocidad baja  (00 en bits OSPEEDR6[1:0])
    GPIOD->PUPDR   &=~ (3<<pinSalida*2); //No pull-up/down (00 en bits PUPDR6 [1:0])
    //Configuración del pin 7 del puerto C (entrada a leer)
    int pinEntrada = 7;
    GPIOC->MODER   &=~ (3<<pinEntrada*2); //Modo ENTRADA    (00 en bits MODER7[1:0])
    GPIOC->PUPDR   &=~ (1<<(pinEntrada*2+1));
    GPIOC->PUPDR   |=  (1<<pinEntrada*2); //Pull-up         (01 en bits PUPDR7 [1:0])

    while (1)
    {
        if(GPIOC->IDR & (1<<pinEntrada))    //Si el estado es alto (reposo)
        {
            GPIOD->BSRRH |= (1<<pinSalida); //Apaga led verde
        }
        else                                //Si el estado es bajo (pulsador presionado)
        {
            GPIOD->BSRRL |= (1<<pinSalida); //Enciende led verde
        }
    }
}
```

Se muestra a continuación una alternativa más práctica para lograr el mismo resultado, utilizando las funciones y etiquetas que aportan las distintas librerías de abstracción de hardware (HAL), con las cuales no es necesario acceder en forma directa a los bits de los registros involucrados.

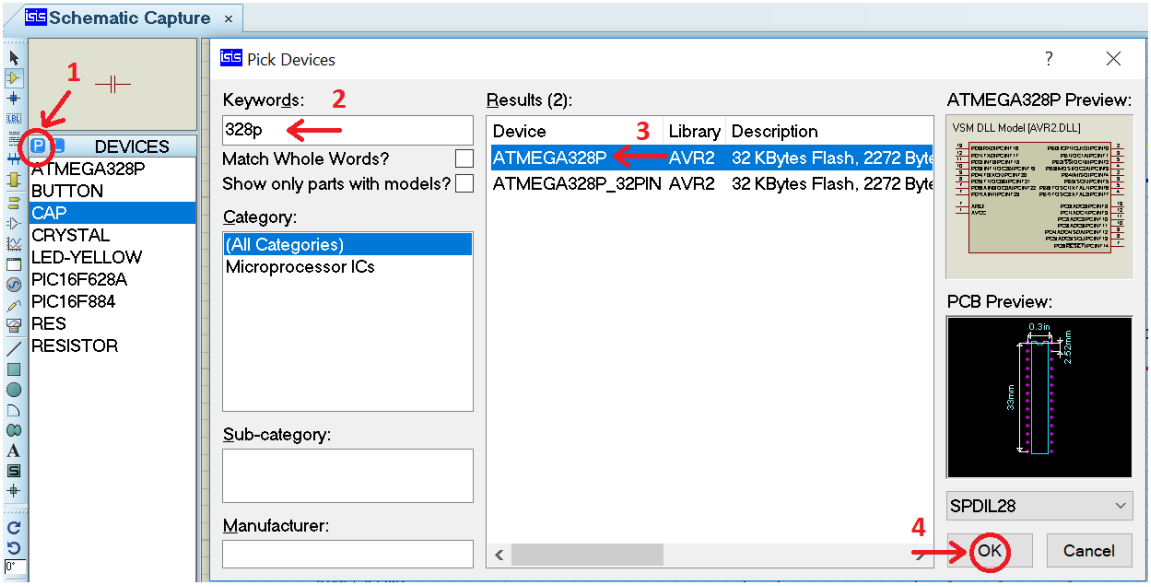
```
#include "stm32f4xx.h"
int main(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;

    //----------PIN PORTD12 en SALIDA (led verde de la placa)-----//
    //Habilitación del clock de GPIO para el puerto C por bus AHB1
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
    //Configuración del pin
    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_12;      //Pin 12
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;    //Salida
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;    //Push-pull
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;     //Sin pull-up ni pull-down
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //Velocidad
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    //----------PIN PORTC7 es ENTRADA (pulsador con pull-up)-----//
    //Habilitación del clock de GPIO para el puerto C por bus AHB1
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    //Configuración del pin
    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_7;       //Pin 7
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN;     //Entrada digital
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;     //Pull-up
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    while (1)
    {
        if(GPIOC->IDR & GPIO_Pin_7)      //Si el estado es alto (reposo)
        {
            GPIOD->BSRRH |= GPIO_Pin_12; //Apaga led verde
        }
        else                             //Si el estado es bajo (pulsador presionado)
        {
            GPIOD->BSRRL |= GPIO_Pin_12; //Enciende led verde
        }
    }
}
```

Se observa la practicidad de modificar fácilmente las etiquetas predefinidas para la configuración del pin a utilizar. Luego se guarda toda la información en una estructura y se llama a la función $GPIO\_Init()$, la cual se encarga de la escritura de bits en los registros del microcontrolador.

Nota: en el entorno Atollic TrueSTUDIO presionando Ctrl+Barra espaciadora nos muestra las opciones de autocompletar para encontrar más rápidamente las etiquetas.

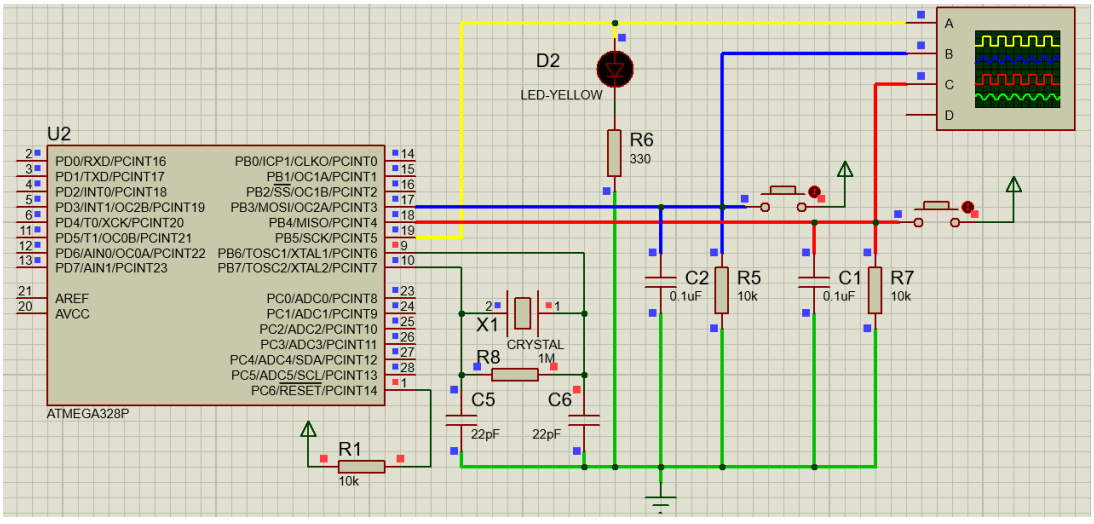
4. Simulación en Proteus

Para verificar el funcionamiento de nuestros programas creamos un nuevo proyecto en *Proteus* (*"File"* → *"New"* → *"Project"*). Luego agregamos los elementos que necesita nuestro circuito. Por ejemplo, para agregar el microcontrolador, seguimos los siguientes pasos:



De igual manera se procede con los demás elementos (resistencias: RES, capacitores: CAP, pulsadores: BUTTON, cristal externo: CRYSTAL, etc.). Además haciendo doble click sobre cada elemento se pueden configurar sus parámetros. Puede ser útil agregar un osciloscopio para ver en tiempo real el estado de los pines del micro, desde el icono instrumentos.

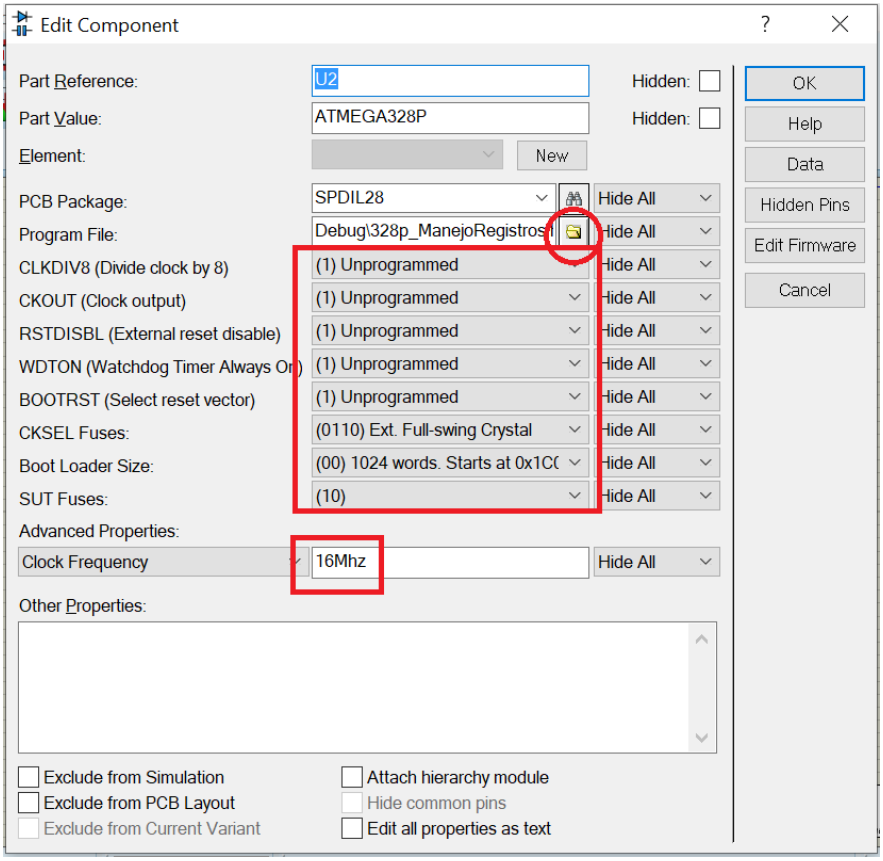
Para el ejemplo básico anterior en Atmega328p, el esquema final queda como sigue:



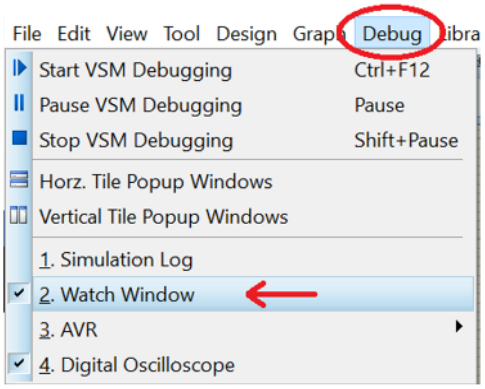
Por último, hacemos doble click sobre el micro y cargamos el programa compilado desde el IDE utilizado, ya sea el archivo .hex para simulación o el archivo .elf (en *AVR*) o .cof (en *PIC*) para simulación y depuración.

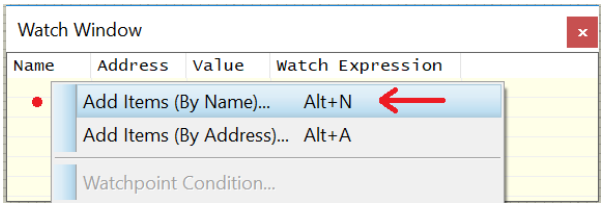
Ver consideraciones prácticas: "Depuración en AVR" [11.9].

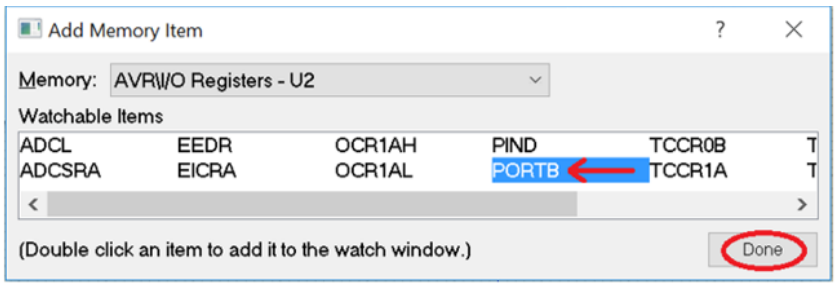
Además, debemos aclarar la frecuencia de trabajo y los fusibles de configuración,como se ve a continuación:



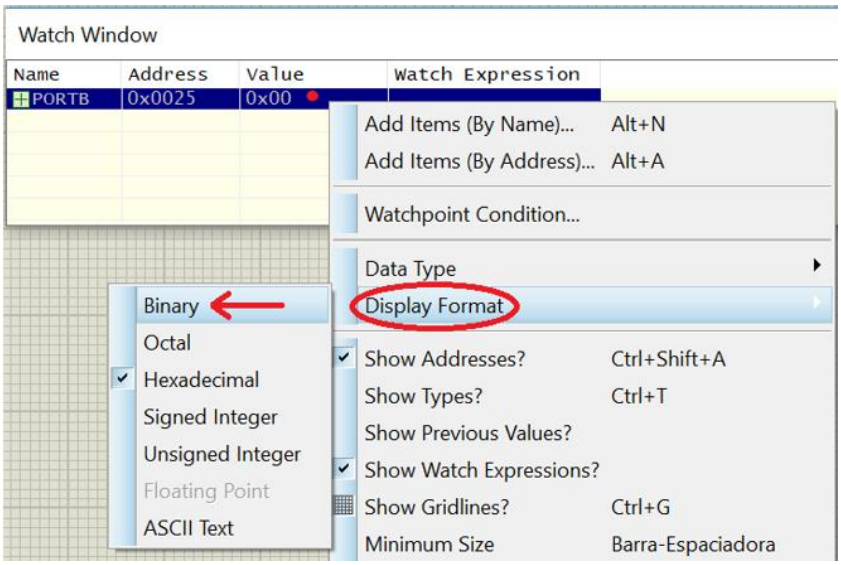
El programa ya está listo para ejecutarse y depurarse. Además, puede ser útil ver la evolución de los bits de los registros internos. Por ejemplo, para observar el PORTB, durante la ejecución del programa vamos a *"Debug"* → *"Watch Window"* → click derecho en la ventana → *"Add Items (By Name)"* → *"PORTB"*.



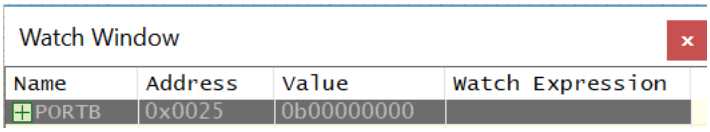




Los valores se muestran por defecto en formato hexadecimal. Para cambiar a formato binario, hacemos click derecho al valor → *"Display Format"* → *"Binary"*.



Así podremos ver la evolución de los bits de este registro, en particular para el ejemplo anterior los bits B3, B4 y B5 que corresponden a los pulsadores y al led.



De manera similar, se puede observar el código fuente, las variables usadas en el programa, memoria de datos, memoria EPROM, etc., desde *"Debug"* → *"AVR"*.

5. Interrupciones

Las interrupciones permiten atender de manera inmediata a los distintos periféricos que pudieran requerirlo (puertos de comunicaciones, timers, puertos paralelos, conversores A/D, etc.). Las interrupciones de periféricos pueden ser habilitadas o enmascaradas por programa, y son disparadas por los eventos que pudiera generar cada periférico. Por ejemplo, el fin de una conversión A/D, la recepción de un byte por puerto serie, un flanco de subida (o bajada) en un pin específico, el cambio de un bit en un puerto, el desborde de un contador interno (timer), etc.

Estos eventos indican mediante un bit denominado **flag** la necesidad de atender o "servir" al periférico en cuestión. Por ejemplo, el **ADIF** (AD interrupt flag) en los microcontroladores con periférico A/D indica que se ha completado una conversión y hay dato listo para ser leído. De manera similar, el **RCIF** (UART Receive interrupt flag) indica que hay un nuevo carácter en el registro de recepción de datos en serie, listo para ser leído. El **INT0IF** (INT0 interrupt flag) indica que se produjo un flanco "activo" en el pin INT0 (de subida o de bajada, según se configure) lo que podría indicar la detección de un sensor externo, etc. El software debe realizar, para el periférico que lo necesite, una **rutina de servicio**, que consiste en código para procesar ese evento.

El software se puede dedicar a verificar periódicamente estos flags (polling), pero esto sobrecarga a la CPU si se realiza con demasiada frecuencia, o se pueden perder eventos si el período de polling es demasiado largo. La idea es que el programa se mantenga en sus tareas habituales y salte a la rutina de servicio del periférico cuando éste lo demande. El software puede selectivamente activar una o más de estas fuentes de interrupciones. Algunas interrupciones especiales son no enmascarables.

5.1. Interrupciones externas e interrupciones por cambio

En ciertas ocasiones es necesario verificar el estado de un pin específico del micro y actuar al detectar un determinado flanco de tensión en el mismo. Por ejemplo, se podría requerir que el ciclo principal del programa realice una determinada tarea A, y que una segunda tarea B se realice cuando en cierto pin ocurra un flanco ascendente, es decir, un cambio de estado bajo (GND) a alto (5V). Dicho cambio de estado puede ser realizado por un pulsador o por otro dispositivo.

Una opción es evaluar la condición del pin (realizar "polling") en cualquier parte del código principal o main del programa. En tal caso, si la evaluación del pin se realiza muy espaciadamente, se corre el riesgo de perder el evento que produjo el flanco ascendente, porque al momento de verificar ya volvió a estado bajo (duración del evento menor al período de "polling"), o podría ser detectado un solo evento cuando en realidad ocurrieron varios (se detecta un flanco sin saber si ocurrieron o no otros flancos entre la verificación actual y la anterior).

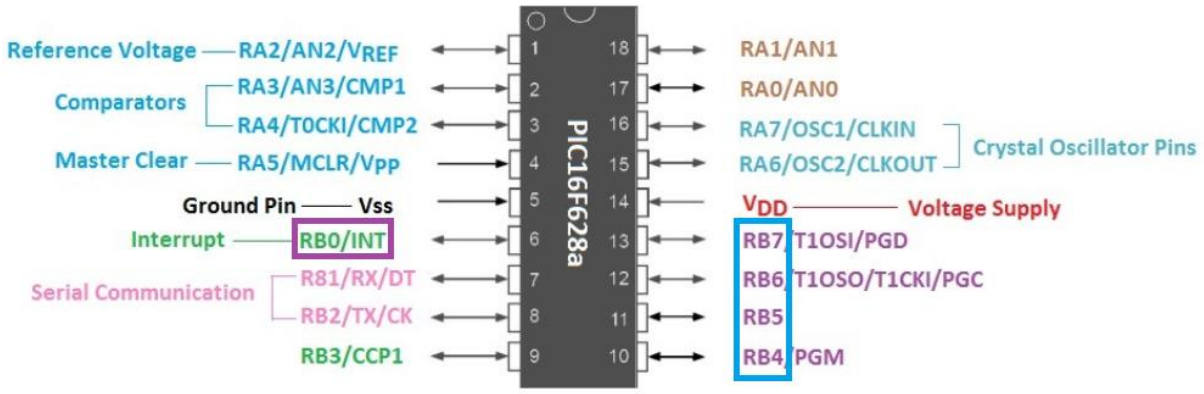
La solución a estos inconvenientes es habilitar las interrupciones externas o por cambio, las cuales son disparadas por la detección inmediata de un ***estado*** o ***cambio de estado*** en ciertos pines asociados a las mismas. Los ***flags*** que levantan estos eventos deben ser borrados por software antes de salir de la ***rutina de servicio*** asociada.

Sin embargo, la interrupción sólo se produce si ésta se habilitó por software (además de la habilitación global) antes de la ocurrencia del evento. En tal caso, al detectar el flag, la tarea que se estaba realizando en el ciclo principal queda momentáneamente detenida y se salta a la rutina de servicio. Una vez finalizada ésta, se retorna al ciclo principal en el mismo punto donde se había detenido.

Importante: para comprender de manera práctica el uso de las interrupciones externas y por cambio, se recomienda ver los ejemplos A, B, C y D, en MyEP2020 [1] → Ejemplos → Interrupciones → AVR → Atmega328p. En estos ejemplos se muestran progresivamente los inconvenientes de realizar polling y las ventajas de habilitar las interrupciones.

5.2. Interrupciones en PIC16F628A

Este micro tiene un pin asociado a ***interrupción externa*** (**RB0/INT**) y cuatro pines asociados a ***interrupción por cambio*** (**RB4:7**). La primera puede elegirse para interrumpir ante un flanco de subida o de bajada. Las segundas interrumpen ante cualquier flanco y están asociadas en un grupo. No se pueden habilitar individualmente, sino que se habilita la interrupción del grupo completo. Todas comparten la misma rutina de servicio, a la cual se entra ante cualquier cambio en uno de esos pines, siempre que sean entradas.



Aunque este micro cuenta con 10 fuentes distintas de interrupción, ante un evento de interrupción, el programa siempre salta a la misma dirección de la memoria (0x0004) y hace polling sobre todos los flags. El orden de verificación da la prioridad. Con la directiva **#priority** se puede establecer dicho orden.

El bit INTEDG del registro ***OPTION_*** permite elegir el tipo de evento que levanta el "flag" de la interrupción externa (1: flanco de subida, 0: flanco de bajada).

OPTION_REG – OPTION REGISTER (ADDRESS: 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

El registro ***INTCON*** contiene los bits de habilitación (1: habilita, 0: deshabilita) y los "flags" de interrupción externa y por cambio.

INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh, 18Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

- **GIE**: habilitación global de interrupción
- **INTE**: habilitación de interrupción externa
- **RBIE**: habilitación de interrupción por cambio
- **INTF**: flag de interrupción externa
- **RBIF**: flag de interrupción por cambio de pines RB4, RB5, RB6 y RB7

Así, por ejemplo, si en nuestro programa queremos realizar tareas inmediatas ante la detección de flancos de subida tanto en el pin **RB0/INT** como en el pin **RB5**, podemos comenzar definiendo los siguientes rótulos auxiliares:

```
#byte PORTB   = 0x06
#byte INTCON  = 0x0B
#byte OPTION_ = 0x81
#bit  RBIF    = INTCON.0
#bit  INTF    = INTCON.1
#bit  T0IF    = INTCON.2
#bit  RBIE    = INTCON.3
#bit  INTE    = INTCON.4
#bit  T0IE    = INTCON.5
#bit  PEIE    = INTCON.6
#bit  GIE     = INTCON.7
#bit  INTEDG  = OPTION_.6
#bit  INT0    = PORTB.0
#bit  RB5     = PORTB.5
```

Habrá además que definir las rutinas de servicio correspondientes, con las tareas que queremos que se ejecuten en las mismas:

```
#int_EXT
void  Rutina_EXT_INT(void)
{
      delay_ms(10);
      if(INT0)     //delay y verificación filtran ruidos de alta frecuencia
      {
         //tarea a realizar
      }
      INTF = 0;
}

#int_RB
void  Rutina_RBIE(void)
{
      delay_ms(10);
      if(RB5)      //delay y verificación filtran ruidos de alta frecuencia
      {
         // tarea a realizar
      }
      RBIF = 0;
}
```

Nota 1: es una buena práctica apagar el flag de cualquier interrupción antes de salir de su rutina de servicio (algunos compiladores lo hacen automáticamente).

Nota 2: como a la segunda rutina de servicio se entra ante cualquier cambio de estado en RB4:7 (que sean entradas), la verificación de estado alto sobre RB5 permite trabajar sólo por flanco de subida en dicho pin, evitando que se realice la tarea dos veces, al pulsar y soltar el pulsador, o ante un cambio en otro pin no deseado.

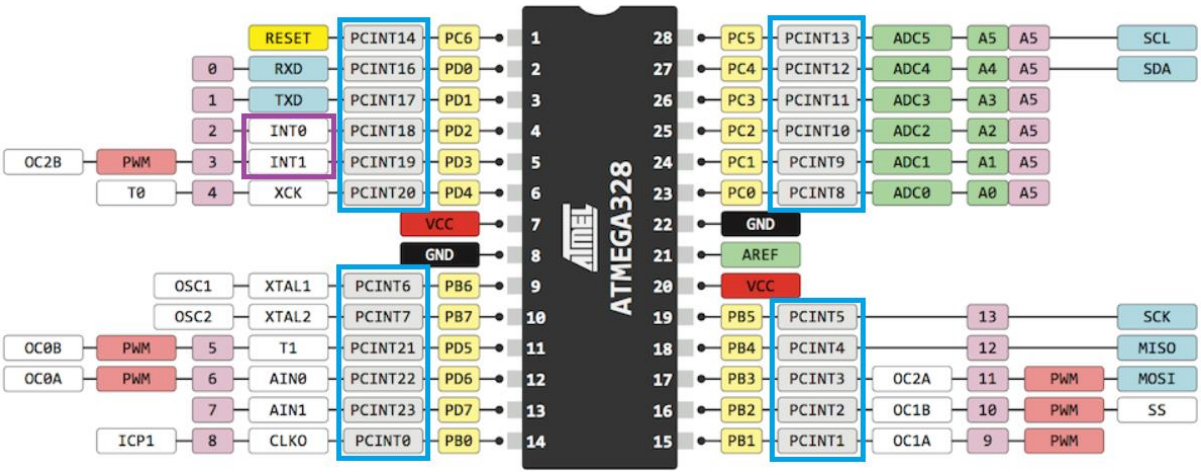
Por último, en la rutina principal del programa habilitamos las interrupciones con las configuraciones deseadas, y activamos la habilitación global de interrupciones:

```
void main()
{
	RBIF   = 0;		//Apaga flag de interrupción por cambio
	INTF   = 0;		//Apaga flag de interrupción por externa
	RBIE   = 1;		//habilita interrupción por cambio
	INTEDG = 1;		//0: flanco de BAJADA - 1:flanco de SUBIDA
	INTE   = 1;		//habilita interrupción externa
	GIE    = 1;		//Habilitación global de interrupciones
	while(TRUE)
	{
		//ciclo infinito
	}
}
```

Nota: es una buena práctica apagar el flag de cualquier interrupción antes de habilitarla.

5.3. Interrupciones en Atmega328p

Este micro tiene dos pines asociados a ***interrupción externa*** (**PD2/INT0** y **PD3/INT1**). El resto de los pines de propósito general están asociados a ***interrupción por cambio*** (**PCINT0:23**). Los pines asociados a las interrupciones pueden ser tanto entradas como salidas. Las primeras pueden elegirse para interrumpir ante un flanco de subida y/o de bajada, o un estado bajo en el pin asociado, cada uno con su rutina de servicio independiente. Las segundas interrumpen ante cualquier flanco y están asociadas en 3 grupos. Aunque se habilitan de manera individual, todas las interrupciones del mismo grupo comparten la misma rutina de servicio.



Se debe tener en cuenta que cada interrupción por cambio activa su respectivo flag al cabo de tres ciclos de reloj (son más lentas que las interrupciones externas). Por otro lado, en este micro las interrupciones están totalmente ***vectorizadas***, es decir, cada evento produce el salto a una posición de memoria distinta, reduciendo el tiempo de atención al evento. Permite además establecer prioridades e interrupciones anidadas.

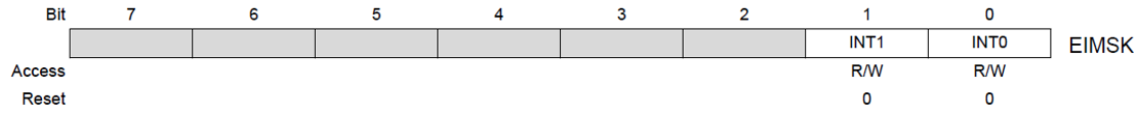
- ***Registros asociados a las interrupciones externas***

El registro ***EICRA*** permite elegir el tipo de **evento** que levanta el "flag" de las interrupciones externas **INTn** (con **n** = **0** ó **1**).

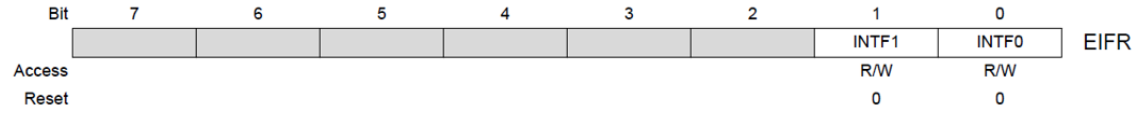
Bit	7	6	5	4	3	2	1	0	
					ISC11	ISC10	ISC01	ISC00	EICRA
Access					R/W	R/W	R/W	R/W	
Reset					0	0	0	0	

ISCn1	ISCn0	Evento que activa flag de INTn
0	0	Nivel bajo
0	1	Cualquier flanco
1	0	Flanco de bajada
1	1	Flanco de subida

El registro ***EIMSK*** permite habilitar las interrupciones externas **INT0** e **INT1** (1 habilita, 0 deshabilita).

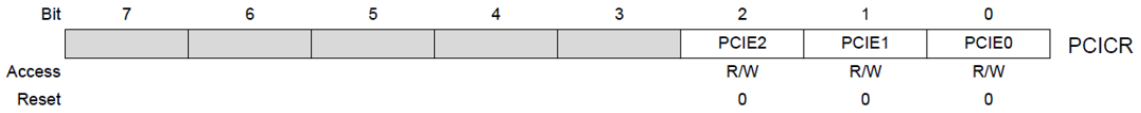


El registro ***EIFR*** contiene a los **flags** de las interrupciones externas **INT0** e **INT1**.



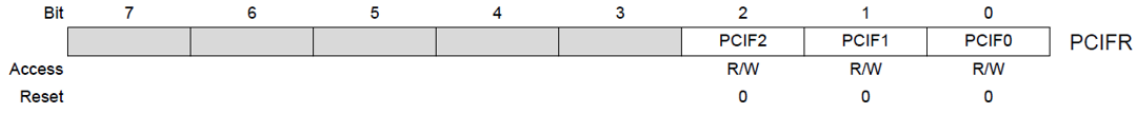
- ***Registros asociados a las interrupciones por cambio***

El registro ***PCICR*** permite habilitar los grupos de pines de interrupción por cambio.



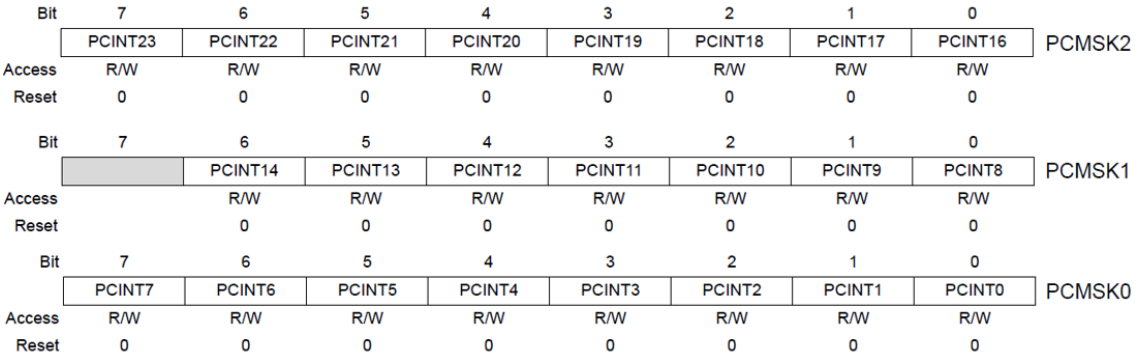
- **PCIE0**: habilita PCINT del grupo de pines PCINT0:7.
- **PCIE1**: habilita PCINT del grupo de pines PCINT8:14.
- **PCIE2**: habilita PCINT del grupo de pines PCINT16:23.

El registro ***PCIFR*** contiene a los flags de los grupos de pines de interrupción por cambio.



- **PCIF0**: flag de PCINT del grupo 0.
- **PCIF1**: flag de PCINT del grupo 1.
- **PCIF2**: flag de PCINT del grupo 2.

Por último, para habilitar la interrupción de un pin específico dentro de su grupo, se pone en '1' su bit correspondiente, de los siguientes tres registros:



Así, por ejemplo, para habilitar la interrupción externa en el pin **PD2/INT0**, será útil definir los siguientes rótulos:

```
#define INT0_ON              (EIMSK |=  (1<<INT0))
#define INT0_OFF             (EIMSK &=~ (1<<INT0))
#define flagINT0_OFF         (EIFR  |=  (1<<INTF0))
#define INT0_test            (PIND   &  (1<<PIND2))
#define INT0_nivelBajo       (EICRA &=~ (3<<ISC00))
#define INT0_cualquierFlanco (EICRA |=  (1<<ISC00))
#define INT0_flancoBajada    (EICRA |=  (2<<ISC00))
#define INT0_flancoSubida    (EICRA |=  (3<<ISC00))
```

Definimos la rutina de servicio asociada, con la tarea que queremos que se ejecute en la misma:

```
//Rutina de servicio interrupcion externa INT0
ISR(INT0_vect)
{
    _delay_ms(10);
    if(INT0_test) //delay y verificación filtran ruidos
    {
        //tarea a realizar
    }
    flagINT0_OFF;
}
```

Nota: es una buena práctica apagar el flag de cualquier interrupción antes de salir de su rutina de servicio (algunos compiladores lo hacen automáticamente).

Luego se puede definir una función auxiliar para habilitar la interrupción con la configuración deseada:

```
void configuraINT0(void)
{
    INT0_flancoSubida;
    flagINT0_OFF;
    INT0_ON;
}
```

Nota: es una buena práctica apagar el flag de cualquier interrupción antes de habilitarla.

Así, en la rutina principal del programa, simplemente habrá que llamar a dicha función y habilitar la interrupción global:

```
int main(void) //rutina principal
{
   configuraINT0();
   sei(); //habilitación global de interrupciones

   while (1)
   {
      // ciclo infinito
   }
   return 0;
}
```

De manera similar podemos habilitar una interrupción por cambio, por ejemplo, en el pin **PD4** (**PCINT20** del grupo 2). Será útil definir los siguientes rótulos:

```
#define PCINT2_ON        (PCICR  |= (1<<PCIE2))
#define flagPCINT2_OFF   (PCIFR  |= (1<<PCIF2))
#define PCINT20_test     (PIND   & (1<<PIND4))
#define PCINT20_ON       (PCMSK2 |= (1<<PCINT20))
```

Definimos la rutina de servicio asociada, con la tarea que queremos que se ejecute en la misma:

```
//Rutina de interrupción por cambio en pines del grupo de PCINT2
ISR(PCINT2_vect)
{
    _delay_ms(10);
    if(PCINT20_test) //delay y verificación filtran ruidos
    {
        //tarea a realizar
    }
    flagPCINT2_OFF; //apaga flag del grupo PCINT2
}
```

Nota 1: es una buena práctica apagar el flag de cualquier interrupción antes de salir de su rutina de servicio (algunos compiladores lo hacen automáticamente).

Nota 2: como estas interrupciones no pueden configurarse para distintos tipos de eventos, sino que entran ante cualquier cambio de estado del pin, se debe verificar si su estado es alto (o bajo) para que actúe por flanco de subida (o bajada), para evitar que entre dos veces a la rutina de servicio al presionar y soltar el pulsador.

Luego se puede definir una función para habilitar la interrupción con la configuración deseada:

```
void configuraInterrupcionPorCambio(void)
{
    flagPCINT2_OFF; //apaga flag del grupo PCINT2
    PCINT2_ON;      //habilita grupo de interrupciones por cambio de pines 16 a 23
    PCINT20_ON;     //habilita interrupción por cambio en PCINT20
}
```

Así, en la rutina principal del programa, simplemente habrá que llamar a dicha función y habilitar la interrupción global:

```
int main(void)
{
   configuraInterrupcionPorCambio();
   sei(); //habilitación global de interrupciones
   while (1)
   {
     // ciclo infinito
   }
   return 0;
}
```

5.4. Interrupciones en Atmega2560

Este micro tiene ocho pines (**INT0:7**) asociados a interrupción externa, de los cuales seis son accesibles desde la placa ArduinoMega (**INT0:5**). Además, al igual que en el Atmega328p, todos los pines de propósito general **PCINT0:23** están asociados a interrupción por cambio, las cuales activan sus respectivos flags al cabo de tres ciclos de reloj. El funcionamiento y los registros asociados son los mismos que en este último. Las únicas diferencias son que ahora los registros asociados a interrupciones externas están completos, y que existen dos registros (en lugar de uno) para elegir el tipo de evento que levanta el "flag" de las interrupciones externas: ***EICRA*** y ***EICRB***.

5.5. Interrupciones en STM32F407

Este micro tiene 23 fuentes de interrupción externa. En la placa Discovery, 16 de las mismas (**EXTI_Line0:15**) están disponibles para direccionarlas a los puertos de entrada/salida, y las 7 restantes están conectadas a periféricos específicos, como se muestra en la siguiente tabla.

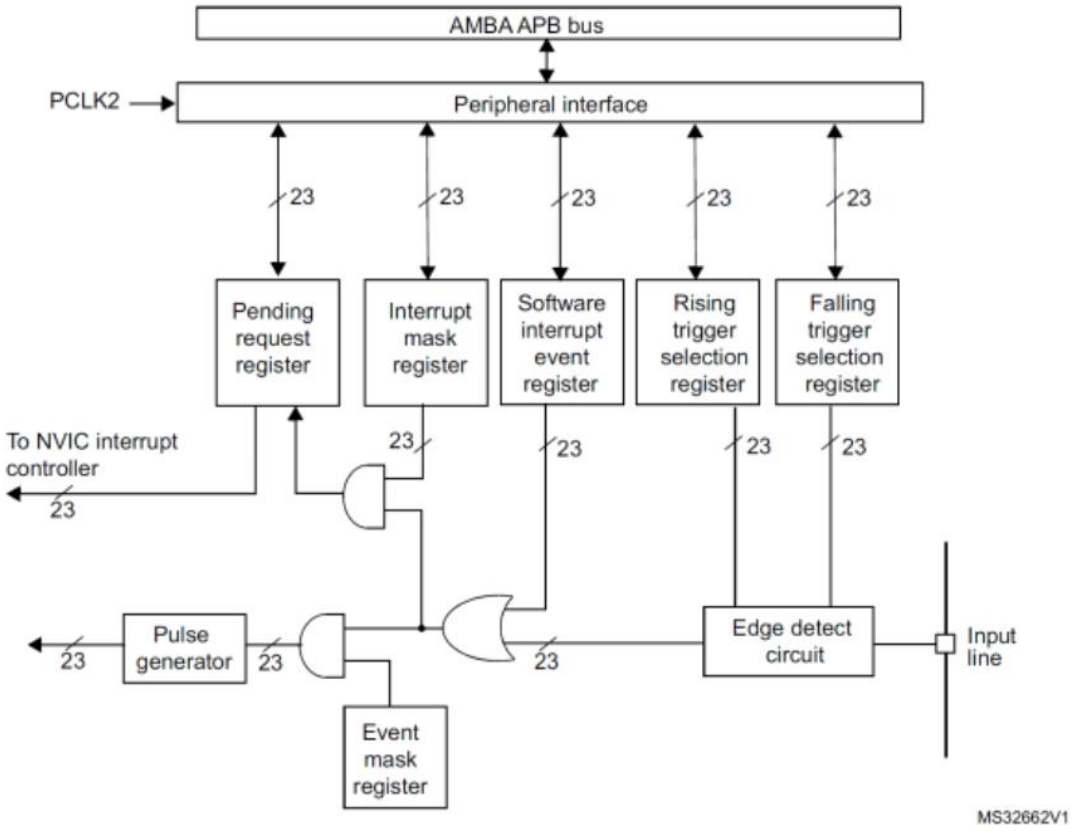
EXTI_Line	Periférico
EXTI_Line16	PVD Output
EXTI_Line17	RTC Alarm event
EXTI_Line18	USB OTG FS Wakeup from suspended event
EXTI_Line19	Ethernet Wakeup event
EXTI_Line20	USB OTG HS Wakeup event
EXTI_Line21	RTC Tramper and Time Stamp events
EXTI_Line22	RTC Wakeup event

Para utilizar una línea de interrupción el pin asociado debe estar configurado como entrada. Luego se debe configurar el manejador de interrupciones (*interrupt handler*) mediante el ***NVIC*** (*nested vector interrupt controller*) que se encarga de seleccionar la interrupción de mayor prioridad (soporta hasta 256 vectores de interrupción diferentes).

En la siguiente tabla se observa el manejador de interrupciones de los pines de I/O para configurar el NVIC:

EXTI_Line	IRQn	ISR
EXTI_Line0	EXTI0_IRQn	EXTI0_IRQHandler
EXTI_Line1	EXTI1_IRQn	EXTI1_IRQHandler
EXTI_Line2	EXTI2_IRQn	EXTI2_IRQHandler
EXTI_Line3	EXTI3_IRQn	EXTI3_IRQHandler
EXTI_Line4	EXTI4_IRQn	EXTI4_IRQHandler
EXTI_Line5 a EXTI_Line9	EXTI9_5_IRQn	EXTI9_5_IRQHandler
EXTI_Line10 a EXTI_Line15	EXTI15_10_IRQn	EXTI15_10_IRQHandler

Los registros ***SYSCFG_EXTICR1:4*** permiten seleccionar le fuente de interrupción. El hardware interno para manejo de interrupciones se ve en la siguiente imagen.

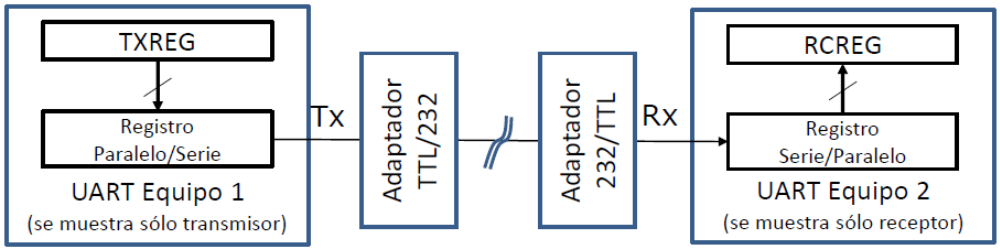


6. Interfaz de comunicación serie asíncrona UART

La UART (Transmisor-Receptor Asíncrono Universal) es un subsistema presente en computadoras, microcontroladores, PLCs, etc., apropiado para comunicar equipos a diversas distancias. Permite transmitir y recibir *caracteres* (normalmente codificados en un byte denominado código ASCII). Sólo existe *línea de datos* entre emisor y receptor (no se requiere línea de "clock" como en interfaces de comunicación síncronas). Para la transmisión a distancia se requiere una adaptación eléctrica según cada norma (*RS-232*, *RS-422* y *RS-485*), a modo de dar robustez a la señal frente a interferencias y ruido.

Los caracteres a *transmitir* por un equipo son cargados en paralelo al registro de transmisión (*TXREG*) de la UART y, mediante un registro de desplazamiento (paralelo/ serie), los bits se presentan en serie en un único pin de salida (*Tx*) como niveles altos y bajos (niveles TTL).

Para la recepción se realiza el proceso contrario. Los bits que llegan en serie ingresan a través de un único pin de entrada (*Rx*) a un registro de desplazamiento (serie/paralelo) y, una vez ingresados todos ellos, son volcados al registro de recepción (*RCREG*) de la UART, de donde puede ser leído el carácter recibido.



Se debe tener en cuenta que un mensaje compuesto por varios caracteres será transmitido como una sucesión de bytes, y estos a su vez como una sucesión de bits. Para delimitar los caracteres se utilizan bits adicionales: un bit de ***START*** ('0') y uno o más bits de ***STOP*** (´1´). Entre caracteres habrá entonces un STOP y luego un START. Tanto antes de comenzar a transmitir como luego del último carácter, la línea queda en silencio, que es un '1'. Por lo tanto, en la práctica una manera de verificar que se ha activado la UART es medir el estado del pin físico Tx, el cual debe estar en alto mientras no se estén transmitiendo datos.

Para que dos equipos puedan comunicarse mediante esta interfaz, la velocidad de envío y recepción de caracteres (***baudrate***) debe ser la misma para ambos (con cierto margen de error admisible).

Ver consideraciones prácticas: "Error de baudrate" [11.4].

6.1. UART en PIC16F628A

La UART de este micro puede configurarse tanto en modo asíncrono (full-duplex) como síncrono (maestro o esclavo, half-duplex). Veremos sólo el modo asíncrono, con transmisión de 8 bits por carácter, 1 bit de STOP, sin paridad, que es lo más habitual. Para empezar, el bit SPEN (RCSTA.7) y los bits TX (TRISB.2) y RX (TRISB.1) deben ponerse en 1 para configurar a los bits RX y TX como los pines de la comunicación. El primer registro asociado a la UART de este *PIC* es el ***TXSTA*** (Registro de control y estado de la Transmisión).

TXSTA – TRANSMIT STATUS AND CONTROL REGISTER (ADDRESS: 98h)

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D
bit 7							bit 0

- **TXEN**: habilitación de pin de transmisión (1: habilitado, 0: deshabilitado)
- **SYNC**: selección de modo de transmisión (1: síncrono, 0: asíncrono)
- **BRGH**: selección de baudrate (X se escribe en el registro ***SPBRG***)
  1: alta velocidad: $\boldsymbol{BAUDRATE = Fosc/(16(X+1))}$
  0: baja velocidad: $\boldsymbol{BAUDRATE = Fosc/(64(X+1))}$
- **TRMR**: estado del registro de desplazamiento de transmisión (1: vacío, 0: lleno)

El segundo registro asociado es el ***RCSTA*** (Registro de control y estado de la Recepción).

RCSTA – RECEIVE STATUS AND CONTROL REGISTER (ADDRESS: 18h)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADEN	FERR	OERR	RX9D
bit 7							bit 0

- **SPEN**: habilitación del puerto serial
- **CREN**: habilitación de recepción continua
- **FERR**: error de frame, por falso bit de STOP
- **OERR**: error de overrun

El registro ***PIE1*** tiene los bits de habilitación de interrupciones por recepción y/o transmisión de datos:

PIE1 – PERIPHERAL INTERRUPT ENABLE REGISTER 1 (ADDRESS: 8Ch)

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
EEIE	CMIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

- **RCIE**: habilitación de interrupción por **recepción** USART
- **TXIE**: habilitación de interrupción por **transmisión** USART

El registro ***PIR1*** contiene los flags de interrupciones por recepción y/o transmisión de datos:

PIR1 – PERIPHERAL INTERRUPT REGISTER 1 (ADDRESS: 0Ch)

R/W-0	R/W-0	R-0	R-0	U-0	R/W-0	R/W-0	R/W-0
EEIF	CMIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

- **RCIF**: flag de interrupción por **recepción** USART
- **TXIF**: flag de interrupción por **transmisión** USART

6.2. USART en Atmega328p

La UART de este micro puede configurarse tanto en modo asíncrono (full-duplex) como síncrono (maestro o esclavo, half-duplex). Veremos sólo el modo asíncrono, con transmisión de 8 bits por caracter, 1 bit de STOP, sin paridad, que es lo más habitual.

Los registros de datos a transmitir y a recibir en este micro comparten la misma dirección, la del registro ***UDR0*** (Registro de Datos de entrada y salida).

UCSR0A: registro 0A de estado y control de la UART:

Bit	7	6	5	4	3	2	1	0	
	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0	UCSR0A
Access	R	R/W	R	R	R	R	R/W	R/W	
Reset	0	0	1	0	0	0	0	0	

- **RXC0** (flag de recepción completa). Este flag se activa cuando hay dato listo para ser leído en el buffer de recepción, y se desactiva cuando este buffer se vacía. Es útil para generar interrupción por recepción de datos serie.
- **TXC0** (flag de transmisión completa).
- **UDRE0** (flag de registro de datos vacío). Se activa cuando **UDRE0** está vacío.

El buffer de transmisión sólo se puede escribir cuando este flag está en 1 (sino el dato será ignorado). Luego se guardará el dato en el registro de desplazamiento de transmisión (cuando el mismo esté vacío) y será enviado en serie por el pin Tx.

- **FE0** (flag de error de frame, por falso bit de STOP).
- **DOR0** (flag de error de overrun).

UCSR0B: registro 0B de estado y control de la UART:

Bit	7	6	5	4	3	2	1	0	
	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80	UCSR0B
Access	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Reset	0	0	0	0	0	0	0	0	

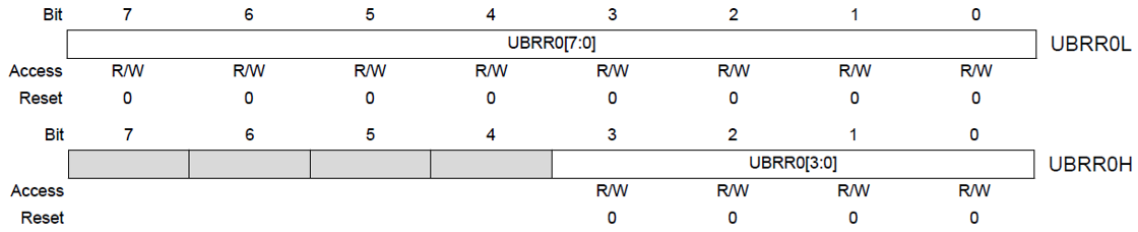
- **RXCIE0**: habilitación de interrupción por recepción completa
- **TXCIE0**: habilitación de interrupción por transmisión completa
- **UDRIE0**: habilitación de interrupción por registro de datos vacío
- **RXEN0**: habilitación de pin RX
- **TXEN0**: habilitación de pin TX
- **UCSZ02**: se usa en combinación con bits del próximo registro (UCSR0C).

UCSR0C: registro 0C de estado y control de la UART:

Bit	7	6	5	4	3	2	1	0	
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 / UDORD0	UCSZ00 / UCPHA0	UCPOL0	UCSR0C
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Reset	0	0	0	0	0	1	1	0	

- **UCSZ01:0**: Junto con el bit UCSZ02 del registro anterior, asignan el **tamaño de carácter** en bits. 011 (valores por defecto) asignan 8 bits.
- **USBS0**: Selección de cantidad de bits de STOP (0: 1 bit, 1: 2 bits).

Para elegir la velocidad de bits por segundo de la transmisión (BAUDRATE) se escribe en los siguientes registros (dirección baja y alta, con resolución de 12 bits.)



$$BAUDRATE = \frac{Fosc}{16(UBRR + 1)}$$

Por lo que, para una velocidad de transmisión deseada, deberá escribirse en este registro:

$$UBRR = \frac{Fosc}{16.BAUDRATE} - 1$$

El buffer de recepción tiene una pila de dos niveles, la cual cambia su estado siempre que el buffer es accedido (cuidado al verificar sus bits).

6.2.1. Ejemplo de USART en Atmega328p

Inicialmente será útil redefinir las primitivas de entrada/salida $fgetc()$ y $fputc()$ que brinda la librería $stdio$, para recibir/transmitir caracteres por la UART correctamente. Así podremos utilizar las funciones $printf()$, $scanf()$, etc. de dicha librería y transmitir números formateados en decimal, hexadecimal, flotante, etc.

```
int mi_putc(char c, FILE *stream)
{
    while(!(UCSR0A&(1<<UDRE0))); // Espera mientras buffer de transmisión esté ocupado
    UDR0 = c;                    // UDR0 recibe el nuevo dato c a transmitir
    return 0;
}

int mi_getc(FILE *stream)
{
    while (!(UCSR0A&(1<<RXC0))); // Espera mientras la recepción no esté completa
    return UDR0;                 // Cuando se completa, se lee UDR0
}

// Redefinimos las primitivas de E/S para recibir/transmitir caracteres por UART
#define fgetc()  mi_getc(&uart_io)
#define fputc(x) mi_putc(x,&uart_io)

// Declara un parámetro tipo stream de E/S para igualar los parámetros en stdio
FILE uart_io = FDEV_SETUP_STREAM(mi_putc, mi_getc, _FDEV_SETUP_RW);
```

A continuación, será útil crear una función genérica para activar la USART0, permitiendo la habilitación de la transmisión y/o recepción de datos, la habilitación de la interrupción por transmisión y/o recepción de datos y la elección de los parámetros de la comunicación (bits de stop, bits de datos, etc.). Por ejemplo:

```
void configuraUART(uint32_t BAUD, uint8_t intRx, uint8_t intTx)
{
    //Parámetros de la comunicación
    UBRR0   =  F_CPU/16/BAUD-1; // Configura baudrate
    UCSR0A &=~ (1<<U2X0);       // Velocidad simple (1 para doble)
    UCSR0B |=  (1<<RXEN0);      // Habilita recepción
    UCSR0B |=  (1<<TXEN0);      // Habilita transmisión
    UCSR0C |=  (1<<USBS0);      // 2 bits de STOP
    UCSR0C |=  (3<<UCSZ00);     // 8 bits de dato
    //El stream (FILE) uart_io es la E/S estandar
    stdout = stdin = &uart_io;
    if(intRx)
    {                           //Apaga flag de interrupción
        UCSR0A |=  (1<<RXC0);   //por Recepción Completa
        UCSR0B |=  (1<<RXCIE0); //Habilita interrupcion RX
    }
    if(intTx)
    {                           //Apaga flag de interrupción
        UCSR0A |=  (1<<TXC0);   //por Transmisión Completa
        UCSR0B |=  (1<<TXCIE0); //Habilita interrupcion TX
    }
}
```

Si se habilitó la interrupción por recepción de datos, habrá que definir su rutina de servicio. Una forma eficiente de manejar la recepción es leer el carácter recibido, almacenarlo en un buffer y salir (de modo que se vuelve a la rutina principal del programa hasta que llegue un nuevo carácter). Luego se interpretará el buffer cuando éste tenga toda la información deseada.

```
unsigned int indcom = 0;    // índice para llenar el buffer de recepción
unsigned int cmd = 0;       // flag "comando en curso"
char comando[30];           // buffer de recepción
ISR(USART_RX_vect)
{
  char dato;
  dato=getc();
  printf(" eco: %c\r\n",dato);
  switch(dato)
  {
    case ':':   // Delimitador de inicio
                indcom=0;   // Inicializa índice de buffer de recepción
                cmd = 1;    // Comando en curso
                break;
    case 8:     // Basckspace
                if(indcom>0) indcom--;
                break;
    case '\r':  // Delimitador de final
                if(cmd==1)              // Si hay comando en curso
                {
                    comando[indcom] = 0; // coloca NULL luego del último caracter
                    interpretaComando(); // Interpreta comando recibido
                    cmd = 0;             // fin de comando en curso
                }
                break;
    default:                            // Todo lo que está entre delimitadores
                if(indcom<30)           // Guarda en elemento del buffer
                    comando[indcom++]=dato;       // e incrementa indcom
                break;
  }
  UCSR0A  |= (1<<RXC0);   //Apaga el flag de interrupción por RX
}
```

Para la implementación de esta rutina de servicio se tuvieron en cuenta las siguientes consideraciones. Cuando se envían comandos por puerto serie, se suele elegir un carácter de inicio (en este caso ':'), una determinada estructura y una carácter de final (en este caso '\r'). Además se debe tener en cuenta que el usuario que transmite puede apretar la tecla backspace, cuyo carácter no hay que almacenar, sino que se debe borrar el último carácter guardado en el buffer. Además, será práctico agregar un carácter *NULL* al final del buffer si lo que se recibe es un comando numérico (permitirá por ejemplo usar la función $atoi()$ que lee hasta *NULL*).

Luego, la función $interpretaComando()$ se deberá adaptar a la estructura de los datos recibidos. Por ejemplo, si se reciben comandos del tipo [:An\r] y [:Bn\r] donde A y B son variables y n es el número que se quiere asignar a las mismas, se puede implementar como sigue:

```
int A, B;
void interpretaComando(void)
{
   switch(comando[0])        // Analiza primer caracter del buffer.
   {
      case 'A':
               if(comando[1])              // si es distinto de 0
               {
                   A = atoi(&comando[1]); // Convierte cadena decimal en entero
                   printf("A = %d\r\n",A);
               }
               break;
      case 'B':
               if(comando[1])              // si es distinto de 0
               {
                   B = atoi(&comando[1]); // Convierte cadena decimal en entero
                   printf("B = %d\r\n",B);
               }
               break;
   }
}
```

Por último, en el programa principal sólo habrá que llamar a la función $configuraUART$ con los parámetros adecuados y, en el caso de haber habilitado las interrupciones por recepción y/o transmisión de datos, habilitar la interrupción global.

```
int  main(void)
{
    configuraUART(myBaudRate,1,0); //(baudrate,intRx,intTx)
    printf("UART OK\r\n");
    sei();         // Habilitación global de Interrupciones
    while(1)
    {
        //bucle principal
    }
}
```

Ver consideraciones prácticas: "Números decimales por puerto serie en AVR" [11.5].

6.3. UART en STM32F407

Este micro tiene 4 USARTs y 2 UARTs, cuyas características se resumen en la tabla 5 del catálogo de este microcontrolador. Al igual que en los ejemplos anteriores, se puede recurrir a las funciones y constantes de las librerías para configurar este periférico. Por ejemplo, para la USART3, los pines TX3 y RX3 son PC10 y PC11, respectivamente. Una función de configuración y habilitación genérica de la misma se muestra a continuación:

```
void UART3_init(int BaudRate, int RxInterr, int TxInterr)
{
  //Estrucutras necesarias para habilitar la USART3
  GPIO_InitTypeDef  GPIO_InitStructure;
  USART_InitTypeDef USART_InitStructure;
  NVIC_InitTypeDef  NVIC_InitStructure;

  //Habilita bus para acceder a USART3
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE);
  GPIO_PinAFConfig(GPIOC, GPIO_PinSource10, GPIO_AF_USART3);
  GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_USART3);

  // Configura pin TX como función alternada
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;

  //Inicia pines Tx y Rx
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
  GPIO_Init(GPIOC, &GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  //Configura UART (baudrate, largo de palabra, bits de stop, paridad, etc.)
  USART_InitStructure.USART_BaudRate   = BaudRate;
  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
  USART_InitStructure.USART_StopBits   = USART_StopBits_1;
  USART_InitStructure.USART_Parity     = USART_Parity_No;
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode       = USART_Mode_Rx | USART_Mode_Tx;

  //Inicializa USART3
  USART_Init(USART3, &USART_InitStructure);

  //Habilita interrupción por recepción de datos (si se requiere)
  if (RxInterr)
  {
   USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);          // habilita Interrupción
   NVIC_InitStructure.NVIC_IRQChannel = USART3_IRQn;       // modificaremos IRQ manualmente
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;// prioridad grupal
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;      // prioridad dentro de grupo
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;         // habilitación global
   NVIC_Init(&NVIC_InitStructure);
  }

  //Habilita USART3
  USART_Cmd(USART3, ENABLE);
}
```

Las funciones para transmitir y recibir caracteres y cadenas se pueden armar como:

```
int USART_puts(USART_TypeDef* UART, char *F)
{
    int c=0;
    while (*F!=0)
    {
      USART_SendData(UART, (uint8_t) *F++);
      while (USART_GetFlagStatus(UART, USART_FLAG_TC) == RESET)
      {}
      c++;
    }
    return c;
}

int USART_putc(USART_TypeDef* UART, char c)
{
      USART_SendData(UART, (uint8_t) c);
      while (USART_GetFlagStatus(UART, USART_FLAG_TC) == RESET)
      {}
      return c;
}

int USART_getc(USART_TypeDef* UARTx)
{
    while(USART_GetFlagStatus(UARTx, USART_IT_RXNE)== RESET)
    {
    }
    int t = UARTx->DR;
    return t;
}
```

Siguiendo los mismos lineamientos para la recepción de datos y almacenamiento en un buffer para su posterior interpretación, vistos en los ejemplos para *PIC* y *AVR*, se puede armar la rutina de servicio de recepción de datos serie como:

```
void USART3_IRQHandler(void)
{
    if( USART_GetITStatus(USART3, USART_IT_RXNE) )
    {
        char dato = USART3->DR;

        switch(dato)
        {
            case ':':   //Delimitador de inicio
                        indcom=0;   // Inicializa índice de buffer de recepción
                        cmd = 1;    // Comando en curso
                        break;
            case 8:     //Backspace
                        if(indcom>0) indcom--;
                        break;
            case '\r':  //Retorno de carro
                        if(cmd==1)
                        {
                            USART_RxStream[indcom]=0; // coloca NULL luego del último caracter
                            interpretaComando();      // Interpreta comando recibido
                            cmd = 0;                  // fin de comando en curso
                        }
                        break;
            default:    //Todos los demás caracteres
                        if(indcom<MAX_STRLEN-1)
                        {
                            USART_RxStream[indcom++]=dato;
                        }
                        break;
        }
    }
}
```

Por último, se muestra la rutina principal del programa:

```
#define UARTN 3
int main(void)
{
    SystemInit(); // Configura PLL etc
    UART_init(UARTN,115200,1,0); // Inicaliza UART y habilita Interrupción de Rx
    sprintf(USART_TxStream,"Probando USART3 de STM32F407\n\r");
    USART_puts(USART3,USART_TxStream);
    while (1)
    {
    }
}
```

7. Temporización

En muchas aplicaciones se requiere un ***control del tiempo*** con determinado nivel de precisión. El objetivo puede ser generar intervalos de tiempo entre acciones o medir el tiempo transcurrido entre eventos. Por ejemplo:

- Generar una secuencia con tiempos específicos.
- Generar pulsos para un driver de motor PaP, siguiendo un perfil de velocidad.
- Generar pulsos repetitivos de ancho preciso para comandar el posicionamiento de un servo de Radio Control, o para control de potencia por PWM.
- Muestrear señales provenientes de sensores (ej.: acelerómetros).
- Realizar un ciclo de control con período preciso (ej.: control PID en tiempo discreto).
- Generar una trayectoria interpolada en un mecanismo multi-eje.
- Registrar datos entre períodos largos de suspensión (data-loggers).
- Medición de tiempos (ej. período, ancho de pulso, tiempo de vuelo en sensor ultrasónico, etc.).

La forma más sencilla de generar retardos entre acciones es mediante funciones de ***delay***, de las cuales disponen los distintos compiladores. Éstas se implementan como ciclos de operaciones neutras (con el único propósito de consumir ciclos).

En ***PIC C Compiler*** se dispone de $delay\_ms()$ y $delay\_us()$, donde el argumento puede ser constante o variable de 16 bits, y de $delay\_cycles()$, donde el argumento debe ser constante de 8 bits.

En **avr-libc** (biblioteca para microcontroladores *AVR*), se dispone de $\_delay\_ms()$ y $\_delay\_us()$, aunque en este caso el argumento puede ser "double" pero solamente constante.

Si bien estas funciones pueden ser precisas, tienen varios inconvenientes:

- Mantienen al procesador ocupado.
- En caso de existir interrupciones se acumulan sus tiempos de atención.
- Se acumula el tiempo de ejecución de otras tareas (muestreo, lectura/escritura de puertos, saltos, incrementos de variables, etc.).

La forma más correcta de realizar un control preciso de tiempo es mediante el uso de ***temporizadores*** o ***timers***. Un microcontrolador básico dispone de al menos un timer, desde 8 hasta 32 bits.

Un ***timer*** es básicamente un contador que se puede leer o escribir, cuya entrada de reloj puede conectarse a distintas fuentes de pulsos. Cuando los pulsos son eventos en un pin de entrada, funciona como un simple ***contador*** (ej. para contar pulsos de un encoder incremental). Cuando los pulsos provienen de un reloj de frecuencia conocida, normalmente el "clock" interno, funciona como ***timer***, porque provee referencia temporal.

Los timers en los microcontroladores cuentan con circuitos accesorios para:

- ***Generar pulsos*** de duración, frecuencia, fase o duty-cycle determinados, en modos denominados "Output Compare y PWM", en una o más salidas que utilicen la misma base de tiempo.
- ***Cronometrar eventos*** en una o más entradas (tiempo entre flancos de subida/bajada, flancos sucesivos, etc.), en modos denominados "Input Capture".

Este conjunto se denomina "Módulo Capture-Compare-PWM" o "Módulo CCP".

Nota: Para saber la velocidad de giro de un eje, lo más preciso es medir el tiempo entre vueltas y calcular su inversa, es decir, obtener la frecuencia a partir del período. En cambio, la obtención directa de la frecuencia (ej. con frecuencímetros) es muy imprecisa para bajas velocidades, o muy lenta (se requieren muchas lecturas para aumentar la precisión).

Ver consideraciones prácticas: "Selección de prescaler, base de tiempo" [11.7].

7.1. Temporización en PIC18F2550

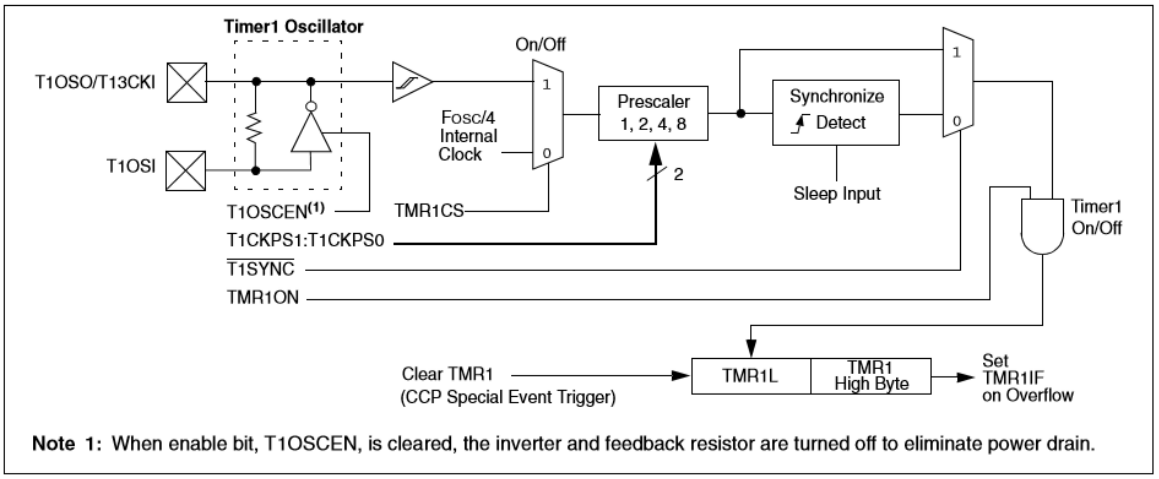
7.1.1. Timer 1 de 16 bits

El ***TIMER1*** del PIC18F2550 incorpora las siguientes características:

- Opción de operación como temporizador o contador de 16 bits.
- Registros de lectura y escritura de 8 bits (TMR1H y TMR1L)
- Opción de fuente de reloj interna o externa. La opción interna a su vez puede oscilador de TIMER1 o reloj del dispositivo.
- Interrupción por desborde de TIMER1 (overflow 0xFFFF → 0x0000).
- Módulo de Reset sobre la generación de un evento especial de CCP (captura, comparación y PWM).
- Flag de estado del dispositivo del reloj (T1RUN).

En la siguiente figura se muestra un diagrama de bloques simplificado del módulo TIMER1. Éste incorpora su propio oscilador de bajo consumo para proveer una opción de reloj adicional. Este timer puede además ser usado para proveer un reloj en tiempo real (RTC) con una mínima adición de código de *overhead* y componentes externos.

FIGURE 12-1: TIMER1 BLOCK DIAGRAM



Uno de los registros de control para este módulo es el T1CON.

REGISTER 12-1: T1CON: TIMER1 CONTROL REGISTER

R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

Con los bits de este registro se puede habilitar el oscilador de TIMER1 (T1OSCEN), habilitar el timer (TMR1ON), seleccionar el prescaler, entre otras configuraciones. Para más detalles ver catálogo del microcontrolador.

Ejemplo de temporización básica:

Se debe generar una interrupción cada 40ms, con un PIC18F2550 que trabaja con cristal externo de 20MHz.

Elegimos vincular al timer con el oscilador del dispositivo. Si la frecuencia de trabajo es de 20MHz, la frecuencia de ciclo es de 20/4=5MHz y el tiempo de ciclo de 0.2us. Si elegimos prescaler de TIMER1 en 1:4, entonces el tiempo entre cada incremento del timer es de 0.8us. Usamos interrupción por desborde de timer, por lo tanto, luego de cada desborde se debe escribir el timer con la diferencia de pulsos entre el valor de desborde (0xFFFF) y los pulsos que deben transcurrir en ese tiempo.

Declaraciones globales en el código, necesarias para configurar este TIMER:

```
#byte   PIE1    = 0xF9D
#bit    TMR1IE  = PIE1.0
#define OVERHEAD 25
#define PULSOS_RESTA 50000      // Tcy=0.2us -> 50000 pulsos para generar 0.01s
```

Configuración del TIMER en la función principal del programa (main):

```
setup_timer_1(T1_INTERNAL|T1_DIV_BY_4);
set_timer1(0xFFFF+OVERHEAD-PULSOS_RESTA);
```

Nota: se debe además habilitar la interrupción de TIMER1 y la interrupción GLOBAL.

Finalmente, se muestra la rutina de servicio de la interrupción por desborde de TIMER1:

```
#int_TIMER1
void  TIMER1_isr(void)
{
     set_timer1(0xFFFF+OVERHEAD-PULSOS_RESTA);
    //tarea a realizar
}
```

Ver consideraciones prácticas: “Overhead” [11.6] y “Selección de prescaler, base de tiempo” [11.7].

7.1.2. Timer 2 de 8 bits

El ***TIMER2*** del PIC18F2550 incorpora las siguientes características:

- Registros de TIMER (***TMR2***) y período (***PR2***), ambos de 8 bits y de escritura/lectura.
- ***Prescaler*** programable por software (1:1, 1:4 y 1:16).
- ***Postscaler*** programable por software (1:1 a 1:16).
- Interrupción por coincidencia de PR2 con TMR2.

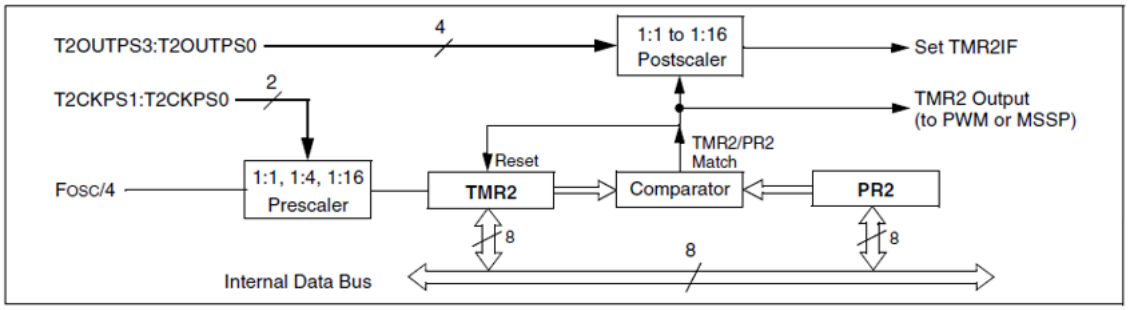
Un registro de control para el TIMER2 es el ***T2CON***, el cual permite habilitar/deshabilitar el timer (TMR2ON), configurar el prescaler (T2CKPS1:0) y el postscaler (T2OUTPS3:0).

REGISTER 13-1: T2CON: TIMER2 CONTROL REGISTER

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

En su operación normal, el TIMER2 se incrementa desde 00h en cada "clock" a una frecuencia igual a Fcy=Fosc/4. El valor de TMR2 se compara con el valor de PR2, en cada ciclo de clock. Cuando ambos valores coinciden, el comparador genera una señal, la cual además resetea el valor de TMR2 a 00h en el próximo ciclo, y maneja el postscaler.

FIGURE 13-1: TIMER2 BLOCK DIAGRAM



Este TIMER puede generar una ***interrupción*** de uso opcional. El contador del postscaler genera un flag de interrupción ante la coincidencia de TMR2 con PR2, el cual encontramos en el bit 1 del registro PIR1 (TMR2IF). Esta interrupción se habilita poniendo en 1 el bit 1 del registro PIE1 (TMR2IE).

TABLE 13-1: REGISTERS ASSOCIATED WITH TIMER2 AS A TIMER/COUNTER

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	51
PIR1	SPPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	54
PIE1	SPPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	54
IPR1	SPPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	54
TMR2	Timer2 Register								52
T2CON	—	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0	52
PR2	Timer2 Period Register								52

Legend: — = unimplemented, read as '0'. Shaded cells are not used by the Timer2 module.
Note 1: These bits are unimplemented on 28-pin devices; always maintain these bits clear.

Se muestra un ejemplo de uso de este módulo en el que se quiere generar un tiempo de muestreo fijo de 20us. Declaraciones globales en el código, necesarias para configurar este TIMER:

```
#byte PR2 = 0xFCB
struct{
      int8  T2CKPS:2;
      int1  TMR2ON;
      int8  T2OUTPS:4;
      int1  nada;
}T2CON;
#locate T2CON = 0xFCA
```

Configuración del TIMER en la función principal del programa (main):

```
PR2 = 99;
T2CON.T2CKPS  = 0b00;   // Prescaler 1:1 (frecuencia igual a Fcy=Fosc/4)
T2CON.T2OUTPS = 0b0000; // Postcaler 1:1 (1 interrupción por coincidencia)
T2CON.TMR2ON  = 1;      // enciende el TIMER2
/* Tcy=0.2us. Poniendo PR2=99, el timer se desborda cada 100*0.2us=20us.
Usando pre y postcaler en 1:1, ocurrirá una interrupción cada 20us. */
```

Finalmente, se muestra la rutina de interrupción por desborde de TIMER2:

```
#int_TIMER2
void  TIMER2_isr(void) //entra cada 20us
{
      //tarea a realizar
}
```

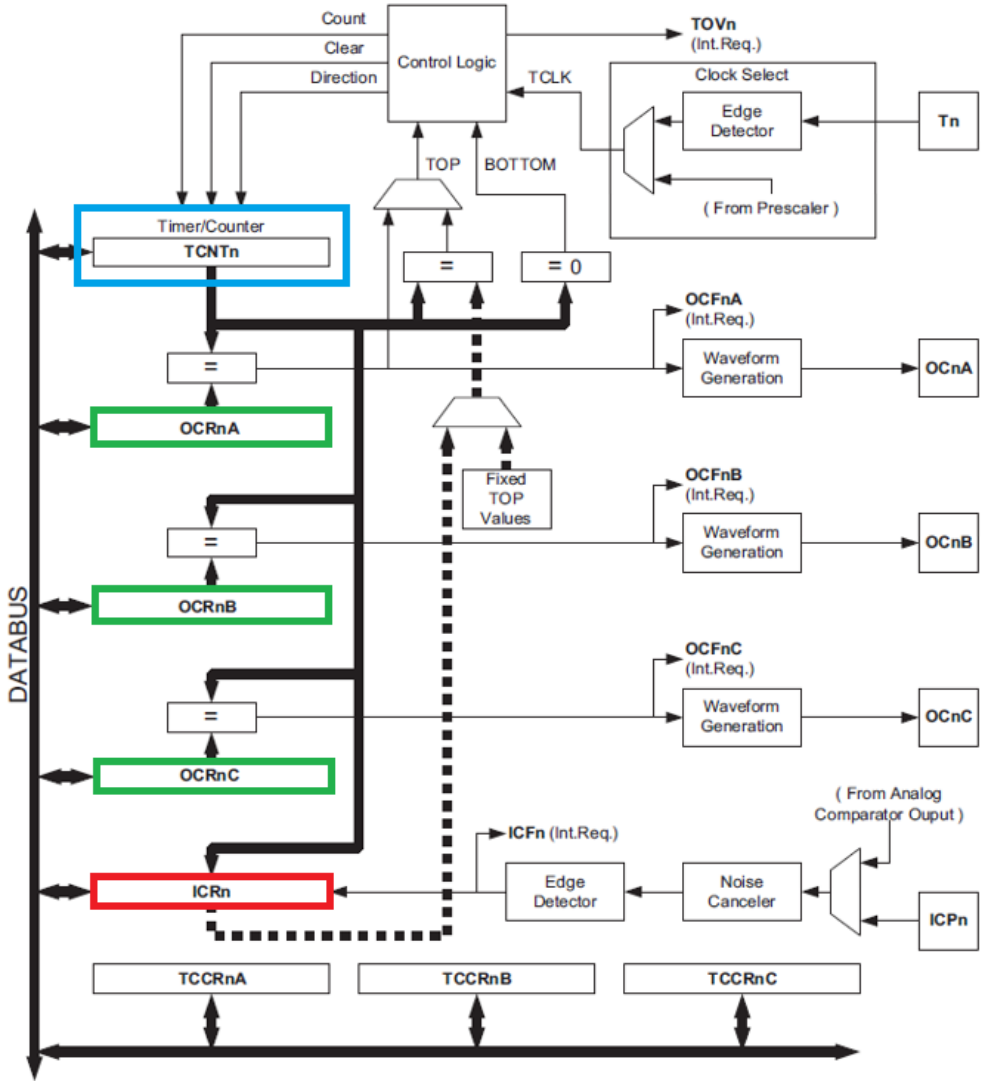
7.2. Temporización en AVR

El ***Atmega328p*** cuenta con dos timers de 8 bits (**TC0** y **TC2**), con prescaler y modo de Comparación, y un timer de 16 bits (**TC1**) con prescaler y modos de Comparación y Captura. Estos permiten controlar de manera precisa los tiempos de ejecución de programa (gestión de eventos), así como también generar y medir ondas o señales. Cuenta además con seis canales de PWM, cuatro de 8 bits (asociados a **TC0** y **TC2**) y dos de 16 bits (asociados a **TC1**).

El ***Atmega2560*** cuenta con dos timers de 8 bits (**TCO** y **TC2**), con prescaler y modo de Comparación, y cuatro timers de 16 bits (**TC1**, **TC3**, **TC4** y **TC5**) con prescaler y modos de Comparación y Captura. Estos permiten controlar de manera precisa los tiempos de ejecución de programa (gestión de eventos), así como también generar y medir ondas o señales. Cuenta además con cuatro canales de PWM de 8 bits (asociados a **TC0** y **TC2**), y 12 canales de PWM con resolución programable de entre 2 y 16 bits (asociados a **TC1**, **TC3**, **TC4** y **TC5**).

7.2.1. Timers de 16 bits

Para los timers **TCn** (con **n**=1 en Atmega328p y **n**=1, 3, 4 y 5 en Atmega2560), es decir, los timers de 16 bits, los registros principales del **TCn** son ***TCNTn***, ***OCRnX*** (con ***X=A, B*** en Atmega328p y ***X=A, B, C*** en Atmega2560) e ***ICRn***, todos de 16 bits.



- El registro ***TCNTn*** contiene el valor del timer en cada instante (se puede escribir y leer).
- Los registros ***OCRnX*** (Output Compare) se escriben con un valor que será comparado permanentemente con el valor de ***TCNTn***. Los ***comparadores*** dan a su salida un '1' sólo en caso de coincidencia o "match" entre sus entradas, y dan un '0' para el resto del tiempo. La coincidencia levanta además un flag (OCnX) que puede usarse para interrupción.
- El registro ***ICRn*** (Input Capture) "congela" el valor de ***TCNTn*** en el momento en el cual se produce el evento configurado en el "Edge detector" (con el bit ICESn del registro TCCRnB) en el pin ICPn. A su vez, en ese momento se levanta un flag de captura ICFn, permitiendo ir a su rutina de servicio (si la correspondiente interrupción está habilitada) a leer el valor "congelado" en ICRn.

Existen distintos **modos de operación** para el **TCn**, los cuales se eligen escribiendo los bits WGM10:3 de los registros ***TCCRnX***.

Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Timer/Counter1 Control Register B

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	

Los 16 **modos de operación** se obtienen con las siguientes combinaciones de los bits WGM:

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Elección del **prescaler** mediante los bits CSn[0:2].

Table 20-7. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
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	clkI/O/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)

Uso de los bits **COMnX**[0:1]:

Table 20-3. Compare Output Mode, non-PWM

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on Compare Match.
1	0	Clear OC1A/OC1B on Compare Match (Set output to low level).
1	1	Set OC1A/OC1B on Compare Match (Set output to high level).

Table 20-4. Compare Output Mode, Fast PWM

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM1[3:0] = 14 or 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode)

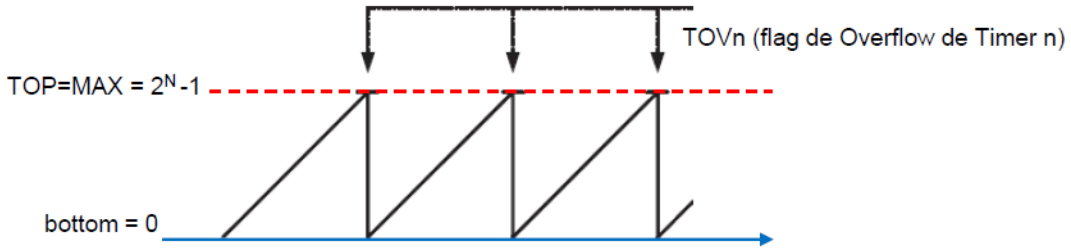
Table 20-5. Compare Output Mode, Phase Correct and Phase and Frequency Correct PWM

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM1[3:0] = 9 or 11: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match when up-counting. Set OC1A/OC1B on Compare Match when down-counting.
1	1	Set OC1A/OC1B on Compare Match when up-counting. Clear OC1A/OC1B on Compare Match when down-counting.

Algunos de los modos de temporización para los timers de 16 bits son los siguientes:

- ***MODO NORMAL (0)***

El timer cuenta desde **0** hasta su valor máximo ($\mathbf{2^{16} - 1}$), desborda y vuelve a **0**, levantando el flag de desborde TOVn. Es decir, desborda cada $\mathbf{2^{16}}$ escalones del timer, cada uno de los cuales se mantiene durante un tiempo $T_{cy} = 1/F_{CPU}$ e inmediatamente se incrementa.



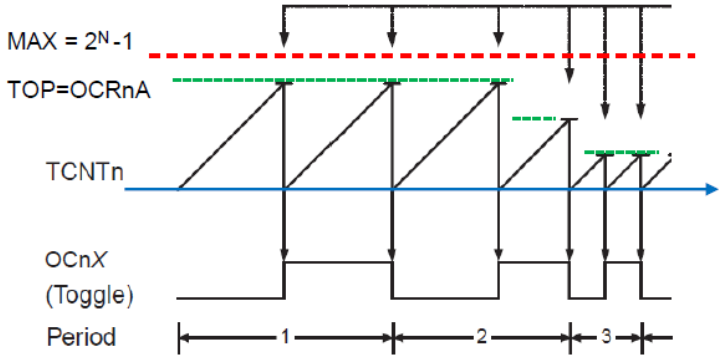
Si se habilita la interrupción por desborde, podemos cambiar el estado de un bit en la rutina de servicio para verificar el tiempo transcurrido. Por ejemplo, para una frecuencia de reloj de 16MHz, con prescaler en 1, el tiempo entre desbordes será de:

$$\boldsymbol{t = \frac{(2^{16})\ ciclos}{(16 * 10^6)\ ciclos/seg} = 4.096ms}$$

Con prescaler en 8, el tiempo será de $\mathbf{8 * 4.096ms = 0.0328s}$, etc.

- ***MODO CLEAR TIMER ON COMPARE MATCH o CTC (4)***

El timer cuenta desde 0 hasta el valor TOP del registro OCRnX, se resetea y levanta el flag de coincidencia OCnX. Puede actuar también sobre el pin OCnX. Para lograr un período de N pulsos (N de 2 a 65536) debe hacerse OCRnX = N-1.



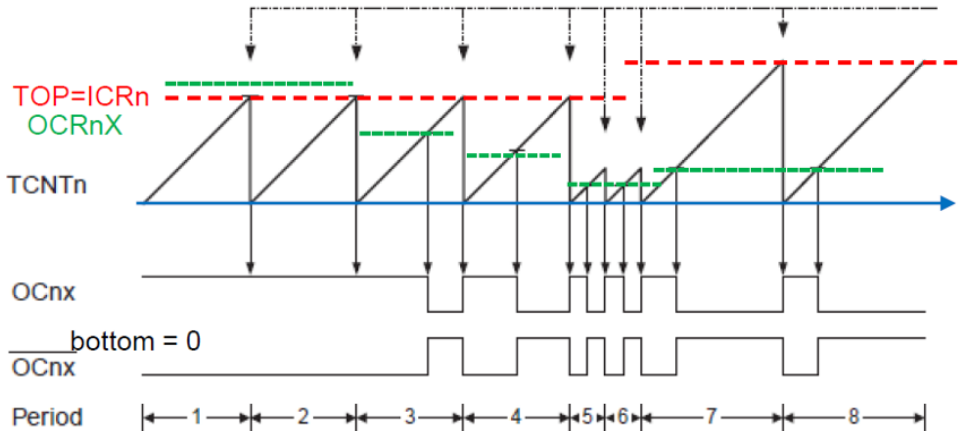
Si se habilita la interrupción por coincidencia, al igual que en el caso anterior, podemos cambiar el estado de un bit en la rutina de servicio para verificar el tiempo transcurrido. Pero una alternativa más fácil es activar el cambio de estado de OCnX en la configuración de los registros. Por ejemplo, para una frecuencia de reloj de 16MHz, con prescaler en 1, si se quiere lograr un tiempo entre coincidencias de 2.5ms, entonces:

$$\boldsymbol{\frac{(OCRnX + 1)\ ciclos}{(16 * 10^6)\ ciclos/seg} = 2.5ms}$$

$$\boldsymbol{OCRnX = 39999}$$

Con esta configuración y prescaler en 256, el tiempo será de $\mathbf{256 * 2.5ms = 0.64s}$, etc.

- ***MODO FAST PWM (14)***



El timer cuenta desde 0 hasta el valor TOP del registro ICRn y se resetea. Además, si la salida OCnX está configurada como "clear", ésta se pone en '1' al comienzo de cada período y se pone en '0' en cada coincidencia del timer con ICRn. Si la misma se configura como "set", actúa en forma opuesta. En definitiva, permite fijar el **período** con **ICRn** y el **duty-cicle** (tiempo en alto) con **OCRnX**.

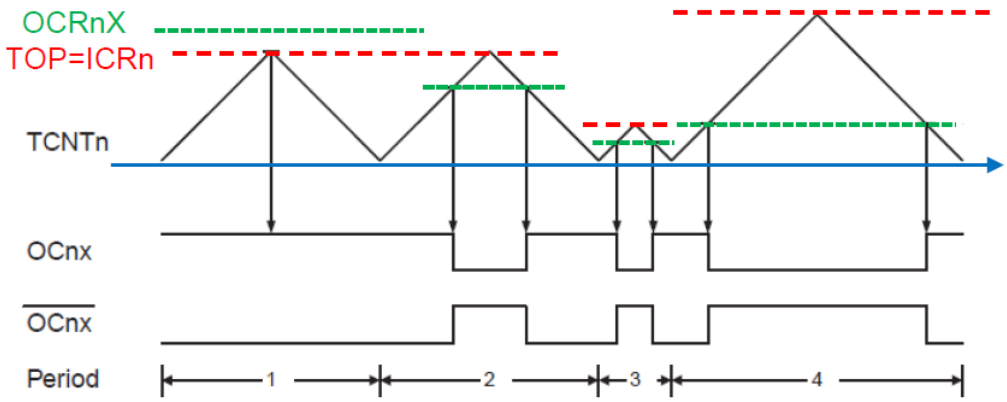
Por ejemplo, para una frecuencia de reloj de 16MHz, con prescaler en 1, si se quiere lograr una señal de PWM al derecho, con un período fijo de 2.5ms, entonces:

$$\frac{(\boldsymbol{ICR1}+\mathbf{1})\ \boldsymbol{ciclos}}{(\mathbf{16}*\mathbf{10^6})\ \boldsymbol{ciclos/seg}} = \mathbf{2.5}\boldsymbol{ms}$$

$$\boldsymbol{ICRn} = \mathbf{39999}$$

Además, la salida de OCnX deberá estar configurada como "clear".

- ***MODO CORRECT PHASE PWM (8)***



El timer cuenta desde 0 hasta el valor TOP del registro ICRn y vuelve en forma descendente hasta 0 nuevamente. Además, la salida OCnX conmuta en cada coincidencia del timer con ICRn. Permite fijar el período con ICRn y el duty-cicle (tiempo en alto) con OCRnX.

7.2.2. Ejemplo de Timer 1 en Atmega328p

En base a la descripción de registros anterior y a las consideraciones prácticas, armaremos un ejemplo de código para implementar distintos modos de uso con Timer1.

En primer lugar, se puede crear una función para selección del prescaler según el tiempo máximo que se desea generar, como sigue:

```
uint16_t prescalerValue_TC1 = 0;
void configuraPrescaler_TC1(float T) //[ms]
{
    T = (float) (T/1000.0);              //pasa a [seg]
    float aux = (float)(pow(2,16)/F_CPU); //máximo valor con prescaler=1
    if     (T<=     aux)    prescalerValue_TC1 = 1;
    else if(T<=   8*aux)    prescalerValue_TC1 = 8;
    else if(T<=  64*aux)    prescalerValue_TC1 = 64;
    else if(T<= 256*aux)    prescalerValue_TC1 = 256;
    else if(T<=1024*aux)    prescalerValue_TC1 = 1024;
    TCCR1B &=~ (7<<CS10); //Resetea a 0 bits de config de prescaler
    switch(prescalerValue_TC1)
    {
        case 0: //timer apagado
                TCCR1B &=~(7<<CS10);
                break;
        case 1:
                TCCR1B |= (1<<CS10);
                break;
        case 8:
                TCCR1B |= (2<<CS10);
                break;
        case 64:
                TCCR1B |= (3<<CS10);
                break;
        case 256:
                TCCR1B |= (4<<CS10);
                break;
        case 1024:
                TCCR1B |= (5<<CS10);
                break;
    }
}
```

Luego se puede crear una función para selección del modo de uso de timer:

```
void configura_Modo_TC1(uint8_t modo)
{
    TCCR1A &=~ (3<<WGM10);
    TCCR1B &=~ (3<<WGM12); //Resetea a 0 bits de config de modo
    switch(modo)
    {
        case 0:
                    TCCR1A &=~ (3<<WGM10);
                    TCCR1B &=~ (3<<WGM12);
                    break;
        case 1:
                    TCCR1A |= (1<<WGM10);
                    break;
        case 2:
                    TCCR1A |= (1<<WGM11);
                    break;
        case 3:
                    TCCR1A |= (3<<WGM10);
                    break;
        case 4:
                    TCCR1B |= (1<<WGM12);
                    break;
        case 5:
                    TCCR1A |= (1<<WGM10);
                    TCCR1B |= (1<<WGM12);
                    break;
        case 6:
                    TCCR1A |= (1<<WGM11);
                    TCCR1B |= (1<<WGM12);
                    break;
        case 7:
                    TCCR1A |= (3<<WGM10);
                    TCCR1B |= (1<<WGM12);
                    break;
        case 8:
                    TCCR1B |= (1<<WGM13);
                    break;
        case 9:
                    TCCR1A |= (1<<WGM10);
                    TCCR1B |= (1<<WGM13);
                    break;
        case 10:
                    TCCR1A |= (1<<WGM11);
                    TCCR1B |= (1<<WGM13);
                    break;
        case 11:
                    TCCR1A |= (3<<WGM10);
                    TCCR1B |= (1<<WGM13);
                    break;
        case 12:
                    TCCR1B |= (3<<WGM12);
                    break;
        case 13:
                    TCCR1A |= (1<<WGM10);
                    TCCR1B |= (3<<WGM12);
                    break;
        case 14:
                    TCCR1A |= (1<<WGM11);
                    TCCR1B |= (3<<WGM12);
                    break;
        case 15:
                    TCCR1A |= (3<<WGM10);
                    TCCR1B |= (3<<WGM12);
                    break;
    }
}
```

A continuación, una función para la selección del tipo de salida de los pines asociados a la temporización, el cual depende del modo de uso del timer. Por ejemplo: modo "clear" y "set" de salidas cuando el modo de uso es PWM, generará salidas de PWM complementarias en los pines asociados. Otro ejemplo: modo "toggle" de salida cuando el modo de uso es CTC, generará una onda cuadrada en el pin asociado.

```
void configura_ModoSalidas_TC1(uint8_t outA, uint8_t outB)
{
    TCCR1A &=~ (3<<COM1A0);
    TCCR1A &=~ (3<<COM1B0);
    switch(outA)
    {
        case 0: //OC1A OFF
                TCCR1A &=~ (3<<COM1A0);
                break;
        case 1: //OC1A toggle
                TCCR1A |= (1<<COM1A0);
                break;
        case 2: //OC1A clear
                TCCR1A |= (1<<COM1A1);
                break;
        case 3: //OC1A set
                TCCR1A |= (3<<COM1A0);
                break;
        default:
                printf("Salida OC1A invalida\r\n");
                break;
    }
    switch(outB)
    {
        case 0: //OC1B OFF
                TCCR1A &=~ (3<<COM1B0);
                break;
        case 1: //OC1B toggle
                TCCR1A |= (1<<COM1B0);
                break;
        case 2: //OC1B clear
                TCCR1A |= (1<<COM1B1);
                break;
        case 3: //OC1B set
                TCCR1A |= (3<<COM1B0);
                break;
        default:
                printf("Salida OC1B invalida\r\n");
                break;
    }
}
```

Nota: los pines asociados a la temporización que se quieran utilizar con algún modo de salida, deben ser declarados como salidas por código antes de habilitar el timer.

Creamos otra función para la habilitación opcional de las distintas fuentes de interrupción disponibles:

```
void interrupciones_TC1(uint8_t InputCapt, uint8_t OutputCaptA,
                        uint8_t OutputCaptB, uint8_t Overflow)
{
    if(InputCapt)
    {
        TIFR1  |=  (1<<ICF1); //apaga flag
        TIMSK1 |= (1<<ICIE1); //habilita interrupción
    }
    if(OutputCaptA)
    {
        TIFR1  |=  (1<<OCF1A); //apaga flag
        TIMSK1 |= (1<<OCIE1A); //habilita interrupción
    }
    if(OutputCaptB)
    {
        TIFR1  |=  (1<<OCF1B); //apaga flag
        TIMSK1 |= (1<<OCIE1B); //habilita interrupción
    }
    if(Overflow)
    {
        TIFR1  |=  (1<<TOV1); //apaga flag
        TIMSK1 |= (1<<TOIE1); //habilita interrupción
    }
}
```

Podemos crear funciones específicas para cada modo de uso de timer. A continuación mostramos sólo ejemplos para los modos 14 y 4:

```
void configura_PWM_TC1(float T, float dutyA, float dutyB)
{
        configura_Modo_TC1(14); //14: fast PWM
        configuraPrescaler_TC1(T);
        ICR1  = (uint16_t)((T/1000)*(F_CPU/prescalerValue_TC1)+1);
        OCR1A = (uint16_t)((dutyA/1000)*(F_CPU/prescalerValue_TC1)+1);
        OCR1B = (uint16_t)((dutyB/1000)*(F_CPU/prescalerValue_TC1)+1);
}

void configura_OndaCuadrada_TC1(float T)
{
        T = T/2;
        configura_Modo_TC1(4);      //4: CTC
        configuraPrescaler_TC1(T);
        OCR1A = (uint16_t)(  ( (T/1000)*(F_CPU/prescalerValue_TC1)+1 ) );
}
```

Finalmente, una función de configuración de timer genérica para llamar en el main principal, la cual llama a las funciones anteriores necesarias. Por ejemplo, para generar ondas cuadradas de período 30ms en pines OC1A y OC1B, podemos escribir:

```
void configuraTIMER1()
 {
        PRR &=~ (1<<PRTIM1); //Reducción de energía para timer 1
        configura_OndaCuadrada_TC1 (30.0); // Período [ms]
        configura_ModoSalidas_TC1(1,1); //(salidaA,salidaB)
        interrupciones_TC1(0,0,0,0); //(InputCap,CompA,CompB,Overflow)
 }
```

7.3. Temporización en STM32F407

Este micro cuenta con un total de 14 timers que permiten controlar de manera precisa los tiempos de ejecución de programa (gestión de eventos), así como también generar y medir ondas o señales. En general, para cada timer ***x*** (x=1:14), el registro ***TIMx_CNT*** contiene el valor del contador, el registro ***TIMx_PSC*** contiene el valor del prescaler (divisor de frecuencia) y el registro ***TIMx_ARR*** contiene el tope o período (valor de desborde). El funcionamiento de los mismos es equivalente al visto en los timers de los micros *PIC* y *AVR*. Una de las diferencias es que el prescaler no está limitado a determinados valores (ejemplo: 0, 1, 8, 64,...etc.), sino que puede tomar cualquier valor entero que permita el tamaño del registro (ejemplo 0 a 65535 para los que tienen el *TIMx_PSC* de 16 bits). Para la generación de PWM, el registro TIMx_ARR determina el período y el registro ***TIMx_CCRn*** (de captura y comparación) determina el duty-cicle (***n*** es el canal usado).

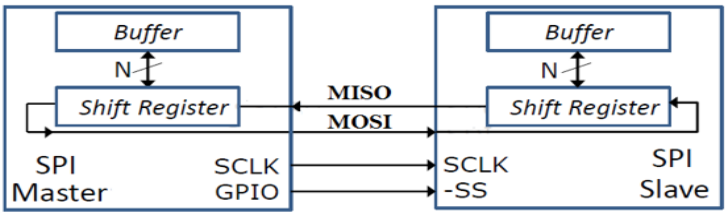
Las características principales de estos timers se resumen en la siguiente tabla:

Timer	Type	Resolution	Prescaler	Channels	MAX INTERFACE CLOCK	MAX TIMER CLOCK*	APB
TIM1, TIM8	Advanced	16bit	16bit	4	SysClk/2	SysClk	2
TIM2, TIM5	General purpose	32bit	16bit	4	SysClk/4	SysClk, SysClk/2	1
TIM3, TIM4	General purpose	16bit	16bit	4	SysClk/4	SysClk, SysClk/2	1
TIM9	General purpose	16bit	16bit	2	SysClk/2	SysClk	2
TIM10, TIM11	General purpose	16bit	16bit	1	SysClk/2	SysClk	2
TIM12	General purpose	16bit	16bit	2	SysClk/4	SysClk, SysClk/2	1
TIM13, TIM14	General purpose	16bit	16bit	1	SysClk/4	SysClk, SysClk/2	1
TIM6, TIM7	Basic	16bit	16bit	0	SysClk/4	SysClk, SysClk/2	1

A grandes rasgos, los timers de propósito general (TIM2:5) tienen incorporado un contador de auto-recarga de 16 o 32 bits, con cuenta creciente y decreciente, prescaler programable de 16 bits para dividir la frecuencia de reloj (por un valor entre 1 y 65535), 4 canales independientes (para captura, comparación, generación de PWM o salida en modo "un pulso") y generación automática de interrupción por distintos eventos. Para más detalles ver catálogo del microcontrolador.

8. Interfaz de comunicación serie síncrona SPI

La **SPI** (serial peripheral interface) es una interfaz de comunicación serie de los microcontroladores, de tipo **síncrona**, **maestro-esclavo**, **full duplex** y de hardware muy simple. Sus puertas de transmisión son de tipo push-pull (complementarias), lo que permite lograr altas velocidades de transmisión, de hasta 10Mbps (mayores a I2C que es de tipo colector abierto). Esta interfaz está presente en prácticamente todos los microcontroladores de gama media-alta y permite conectarse con una gran variedad de periféricos. Además, en aquellos micros sin interfaz SPI de hardware, es muy sencillo implementarla por software.



El maestro es el que maneja la señal de reloj SCLK (serial clock). Además, maneja su salida de datos SDO o MOSI (Master Out, Slave In), que ingresará a los esclavos. Mediante un pin de propósito general cualquiera GPIO del maestro configurado como salida, se maneja la señal de selección de esclavo –SS (Slave Select), la cual se activa usualmente con '0'. Cuando –SS de un esclavo está en '1' (inactiva), el mismo es insensible a las señales en sus entradas (SCLK y MOSI) y mantiene su salida SDO o MISO (Master In, Slave Out) en tercer estado. Esto permite una conexión en bus de varios esclavos comandados por un maestro.

Internamente hay un único registro de desplazamiento (tanto para transmisión como para recepción) para cada dispositivo. Estos son normalmente de 8 bits (también hay de 16 o más). Si se ha puesto el -SS del esclavo en 0, al escribir el buffer del maestro con un dato de N bits, éste se vuelca automáticamente en su registro de desplazamiento y se traslada bit a bit hacia el esclavo, por cada pulso de SCLK que envíe el maestro. Al completarse los N pulsos de SLCK, con el dato contenido en el registro de desplazamiento del esclavo, éste se vuelca en su buffer. Si se envían N pulsos más, el dato vuelve al registro de desplazamiento del maestro, ya que las líneas están conectadas en anillo.

8.1. Interfaz SPI en Atmega328p

Este micro dispone de dos interfaces SPI maestro-esclavo. Una de ellas es dedicada y la otra es la USART, que puede utilizarse en modo síncrono.

El primer registro asociado es el ***SPCR***:

7	6	5	4	3	2	1	0	
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

- ***SPIE***: habilitación de interrupción SPI
- ***SPE***: habilitación de la interfaz SPI
- ***DORD***: transmisión del dato desde el bit más o menos significativo (0: bit más significativo primero, 1: bit menos significativo primero)
- ***MSTR***: modo SPI (1: maestro, 0: esclavo)

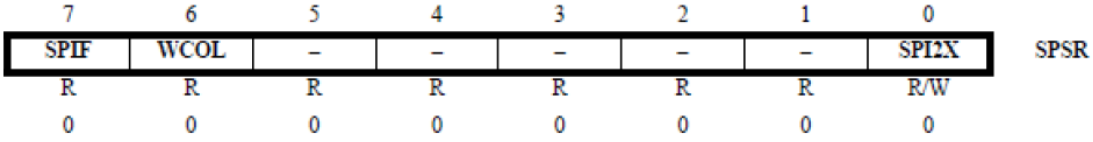
- ***CPOL*** y ***CPHA***: modo de funcionamiento (estado de SLCK en reposo, y flanco de SCLK para lectura de bit).

Modo	SCLK en reposo	SCLK en lectura de bits
0	bajo	flanco subida
1	bajo	flanco bajada
2	alto	flanco bajada
3	alto	flanco subida

- ***SPR1*** y ***SPR0***: selección de prescaler.

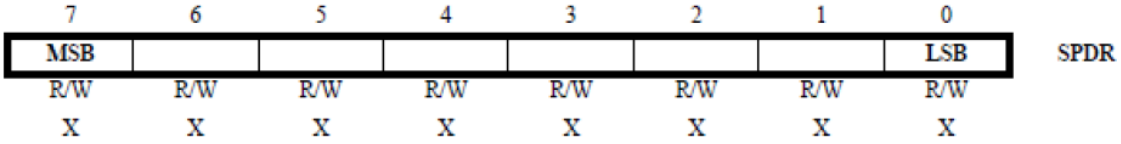
SPR1:SPR0	prescaler
0	4
1	16
2	64
3	128

El siguiente registro asociado es el ***SPSR***:



- **SPIF**: flag de interrupción.
- **WCOL**: flags de detección de colisión.
- **SPI2X**: duplica la velocidad, para cualquier prescaler elegido.

Por último, el registro ***SPDR***, en el cual se puede escribir un dato o leer el dato que contiene.



Nota 1: Luego de desplazarse un byte completo, el generador de pulsos de SCLK se detiene y se activa el flag de fin de transmisión de SPI (SPIF). Si está activado el bit de interrupción de SPI (SPIE) del registro SPCR, esto produce una interrupción.

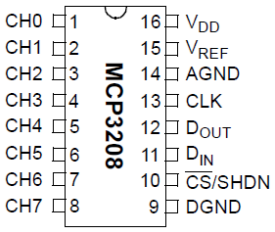
Nota 2: Para asegurar el correcto muestreo de la señal de reloj, los períodos mínimos en estado bajo y alto deben ser mayores a dos ciclos de CPU.

Con este conocimiento se puede armar un código genérico para activar la interfaz SPI para comunicarse con uno o varios dispositivos, eligiendo el modo de funcionamiento y el prescaler que da la velocidad de transmisión. Se muestran además etiquetas que serán útiles en las distintas partes del programa:

```
#define MOSI_OUT (DDRB |=  (1<<DDB3))
#define MISO_IN  (DDRB &=~ (1<<DDB4))
#define SCLK_OUT (DDRB |=  (1<<DDB5))
#define CS_OFF   (PORTB &=~ (1<<PORTB1))
#define CS_ON    (PORTB |=  (1<<PORTB1))
#define CS_OUT   (DDRB  |=  (1<<PORTB1))

#define SPI_ON         (SPCR |=  (1<<SPE) )     //habilita interfaz SPI
#define SPI_OFF        (SPCR &=~ (1<<SPE) )     //deshabilita interfaz SPI
#define modo_Master    (SPCR |=  (1<<MSTR))     //modo maestro
#define modo_Slave     (SPCR &=~ (1<<MSTR))     //modo esclavo
#define FinTransmision (SPSR  &  (1<<SPIF))

void configuraSPI_Master(uint8_t modo, uint8_t prescaler)
{                     //(modo: 0,1,2,3, prescaler:4,16,64,128)
    switch(modo)
    {
        case 0:
                SPCR &=~ (3<<CPHA);
                break;
        case 1:
                SPCR |=  (1<<CPHA);
                break;
        case 2:
                SPCR |=  (1<<CPOL);
                break;
        case 3:
                SPCR |=  (3<<CPHA);
                break;
        default:
                printf("Modo no valido\r\n");
                break;
    }
    switch(prescaler)
    {
        case 4:
                SPCR &=~ (3<<SPR0);
                break;
        case 16:
                SPCR |=  (1<<SPR0);
                break;
        case 64:
                SPCR |=  (1<<SPR1);
                break;
        case 128:
                SPCR |=  (3<<SPR0);
                break;
        default:
                printf("Prescaler no valido\r\n");
                break;
    }
    SPI_ON; modo_Master;
}
```

8.1.1. Ejemplo de aplicación: lectura de conversor A/D MCP3208

El MCP3208 es un conversor Analógico/Digital de 12 bits de resolución, que cuenta con 8 canales independientes de entradas analógicas, las cuales son transformadas a números digitales para trasmitirlos mediante interfaz SPI. La comunicación con este dispositivo se inicia poniendo en estado bajo a su pin CS (el mismo debe comenzar en estado alto, y volver a estado alto entre conversiones).

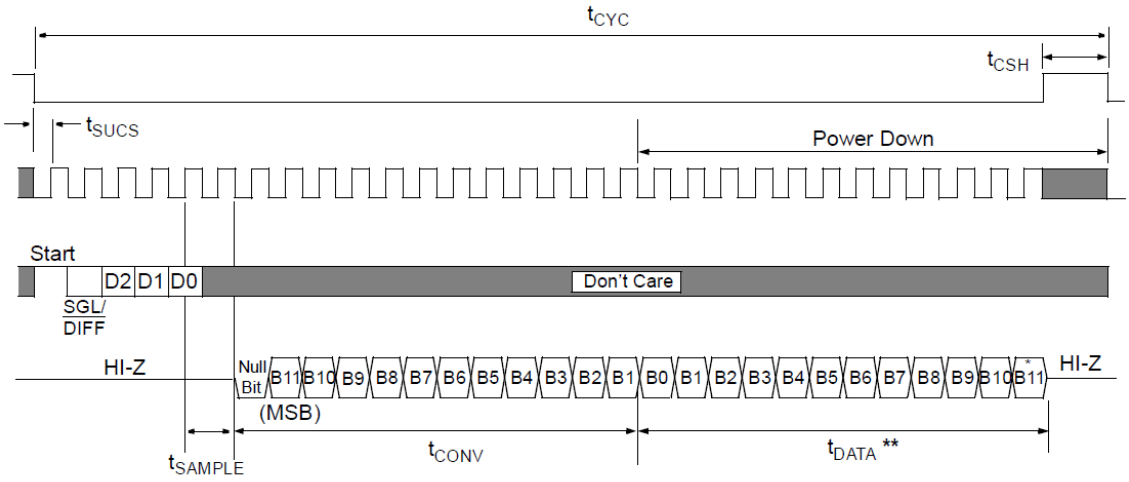


El primer flanco en el pin SCLK con CS en estado bajo y Din en cualquier valor positivo, significa un bit de START. Luego de este bit, le siguen los bits **S – D2 – D1 – D0**. Donde:

- S → 1: canal individual, 0: diferencia entre canales consecutivos (0 y 1, 2 y 3…).
- D2-D1-D0 → canal analógico a leer (0 a 8, en binario).

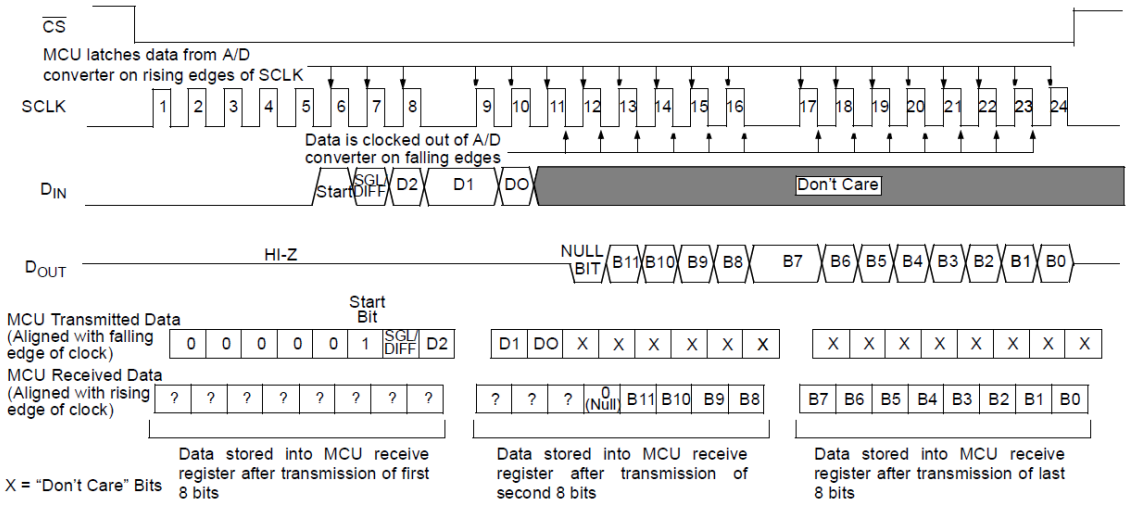
El dispositivo comenzará a leer la entrada analógica en el canal especificado en el cuarto flanco ascendente en SCLK luego del bit de START, y finalizará la lectura en el quinto flanco descendente en SLCK luego del bit de START.

Una vez que entró D0, se requiere un ciclo de SCLK más para completar el período de "Sample and Hold". En el flanco descendente del próximo ciclo, el dispositivo pondrá en su salida un bit nulo, y los próximos 10 ciclos llevarán el resultado de la conversión con el bit más significativo primero. Si se siguen enviando ciclos de SCLK con CS en estado bajo, luego de 10 ciclos se devolverá el resultado con el bit menos significativo primero. Si en este momento se siguen mandando pulsos de SCLK con CS en estado bajo, el dispositivo devuelve ceros.



Communication with MCP3204 or MCP3208 in LSB First Format.

Sin embargo, como sólo de pueden enviar grupos de 8 bits desde un microcontrolador, se tienen que enviar más pulsos de SCLK de los requeridos normalmente. Esto se pueden realizar enviando ceros de relleno antes del bit de START, como se muestra a continuación:



SPI Communication using 8-bit segments (Mode 0,0: SCLK idles low).

Observando las dos últimas líneas de datos de este esquema, se puede armar una función genérica para leer canales de este dispositivo, como sigue:

```
uint16_t ReadADC(uint8_t ch)
{
    uint8_t aux,data_high,data_low;
    aux=0x06;
    if(ch>3) aux|=0x01;
    CS_OFF;
    SPI_Master_Transfer(aux); //devuelve 8 bits que no importan
    aux = (0xFF & ch<<6 );
    data_high=SPI_Master_Transfer(aux);
    data_high&=0b00001111;
    data_low=SPI_Master_Transfer(0xDC); //cualquier byte
    _delay_us(2);
    CS_ON;
    return ((data_high<<8)|data_low);
}
```

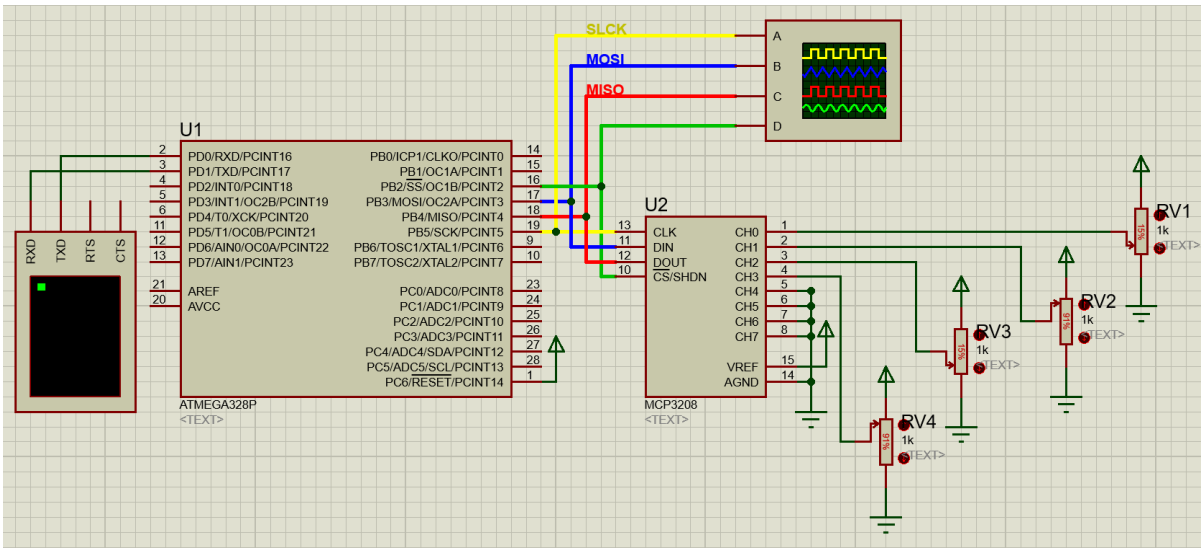
La función de transferencia de datos del maestro se armó considerando que, luego de leer el registro de datos SPDR, se levanta el flag SPIF del registro SPSR al finalizar la transmisión.

```
char SPI_Master_Transfer(uint8_t dato)
{
     SPDR = dato;            //Escribe dato en registro SPDR
     while(SPSR&(1<<SPIF));  //Espera hasta que finalice la transmisión
     return SPDR;            //Devuelve registro
}
```

Finalmente, un ejemplo de código principal para leer 4 canales analógicos en forma cíclica es el siguiente:

```c
int main(void)
{
    configuraUART(myBaudRate,0,0);
    configuraSPI_Master(0,128);
    //(Modo,prescaler) -> modo 0, SCK=16MHz/128=125kbps

    CS_ON; CS_OUT; MOSI_OUT; SCLK_OUT; MISO_IN;

    uint16_t data;
    while (1)    //Lee 4 canales analógicos y muestra su valor por la UART
    {
        for(uint8_t canal=0; canal<4; canal++)
        {
            data = ReadADC(canal);
            printf("%u ",data);
        } printf("\r\n");
        _delay_ms(300);
    }
}
```

Esquemático:



9. Interfaz de comunicación serie síncrona I2C

En las interfaces **síncronas** de comunicación serie, entre emisor y receptor hay dos líneas, una de datos y otra de reloj. Puede haber desfasaje en las velocidades de propagación de estas señales cuando la distancia es grande, por lo que son interfaces apropiadas para comunicar circuitos integrados dentro de un mismo equipo (distancias cortas). Además, no requieren adaptación eléctrica, excepto alguna resistencia de "pull-up" o para interconectar circuitos integrados de distintos voltajes de alimentación (por ejemplo, para conectar un sensor MPU6050 de 3.3V con un microcontrolador de 5V).

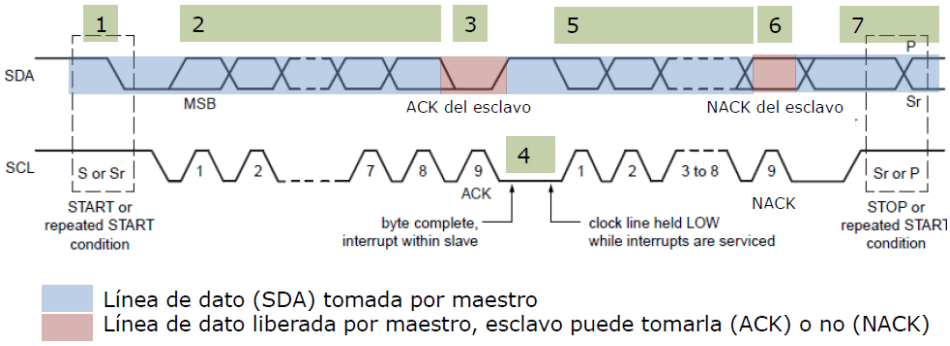
En particular I^2C es una interfaz de comunicación **serie síncrona**, **bidireccional, maestro-esclavo** o **maestro flotante**, con una línea de datos ***SDA*** (bidireccional) y una línea de reloj ***SCL*** (unidireccional, de maestro a esclavo). Es de tipo "colector abierto", con resistencias de pull-up en cada línea, a diferencia de la interfaz ***SPI*** que trabaja a "salida complementaria". La salida complementaria permite mayores velocidades de transmisión, pero existe peligro de colisión eléctrica cuando dos dispositivos intentan transmitir al mismo tiempo, provocando un cortocircuito que puede dañarlos.

Ver consideraciones prácticas: "Salida complementaria vs colector abierto" [11.8].

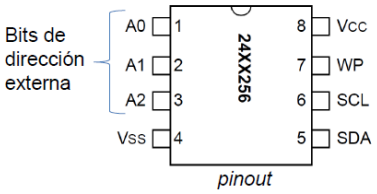
En el caso de un sistema maestro-esclavo, un único dispositivo (maestro) maneja la línea de reloj, iniciando ya sea una escritura o una lectura de parámetros. Las ***transacciones*** comienzan con un bit de START (flanco de bajada en SDA mientras SCL='1') y terminan con un bit de STOP (flanco de subida en SDA mientras SCL='1'). Ambos bits son generados por el maestro, y son los únicos cambiantes durante SCL='1'. Los demás bits en SDA deben permanecer estables durante SCL='1' para que no sean interpretados como START o STOP.

Las transacciones son iniciadas por el maestro. El primer byte luego de un START contiene los bits de direccionamiento del esclavo más un bit adicional (0 o 1 si lo que sigue es una escritura o una lectura, respectivamente). Los siguientes bytes dependen de la operación a realizar, y son específicos del periférico con el que se esté comunicando. Hasta el próximo START, los demás esclavos (si los hubiera), ignoran los mensajes.

Luego de enviar cada byte, el maestro libera la línea SDA (la deja en alto) y espera hasta que el esclavo responda bajando la línea, para asegurarse de que ha recibido bien la información, lo que se conoce como ACK (acknowledgment). Durante esa espera, el esclavo toma la línea de SCL dejándola en '0', y la libera cuando está listo. Ambas acciones son posibles ya que SCL y SDA son salidas tipo colector abierto (el '0' es dominante).



9.1. Ejemplo de aplicación: escritura/lectura de memoria 24LC256



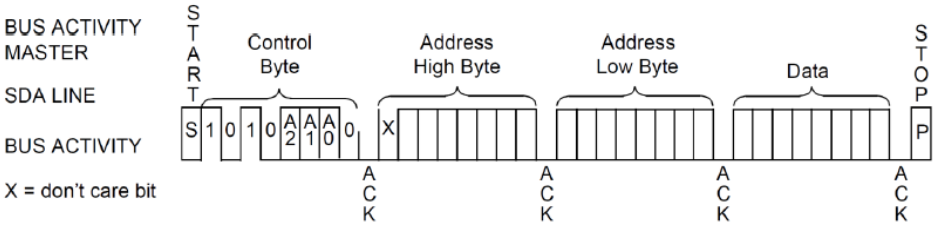
pinout

- Memoria Serial EEPROM de 256 kbits, organizados en 32 kBytes.
- 1.000.000 de ciclos de borrado/escritura
- Hasta 8 chips direccionables (por A2, A1, A0)
- Transferencia I2C hasta 400kHz

Estas memorias tienen una dirección estándar 1010xxx, por lo tanto, los primeros 4 de los 7 bits de dirección (de izquierda a derecha) son "1010". Los 3 bits restantes se corresponden con los 3 pines A2 A1 A0. Es decir, una EEPROM que tenga sus pines A2, A1, A0 a masa tendrá la dirección 1010000, y si todos están a Vcc responderá a la dirección 1010111.

9.1.1. Escritura de un byte

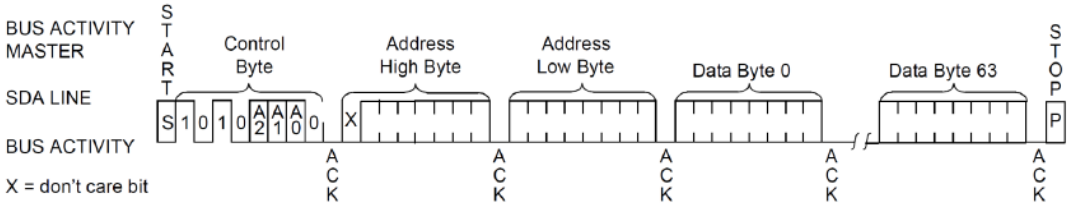
Para escribir un byte de dato en una dirección determinada, se envía un comando de la forma:



9.1.2. Escritura secuencial

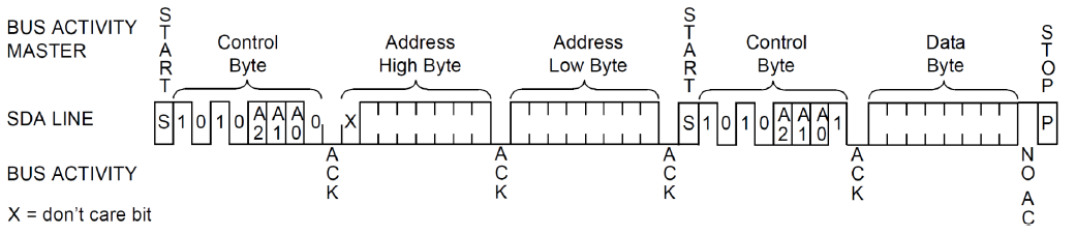
La escritura directa en EEPROM inicia cuando se envía el bit de STOP, y demora hasta 5ms ya sea para uno o múltiples bytes. Además, mientras se realiza este proceso la EEPROM no puede recibir ni transmitir datos. Para acelerar el proceso las EEPROM serie tienen un buffer RAM, lo que permite realizar una escritura secuencial de múltiples bytes, para luego volcarlos a la EEPROM en una única operación. En la memoria 24LC256, este buffer es de 64 bytes. Si se sobrepasa la capacidad de la página, se sobre escribe lo anterior (roll-over).

Si se envía START y el byte de control mientras la memoria no está lista, ésta responde con NACK (nivel alto en SDA). Para evitar pérdida de datos se puede esperar más de 5ms (por ejemplo, mediante un delay) entre cada bit de STOP y una nueva operación. Otra forma más efectiva es enviar START y el byte de control hasta que se detecte el ACK (EEPROM lista).



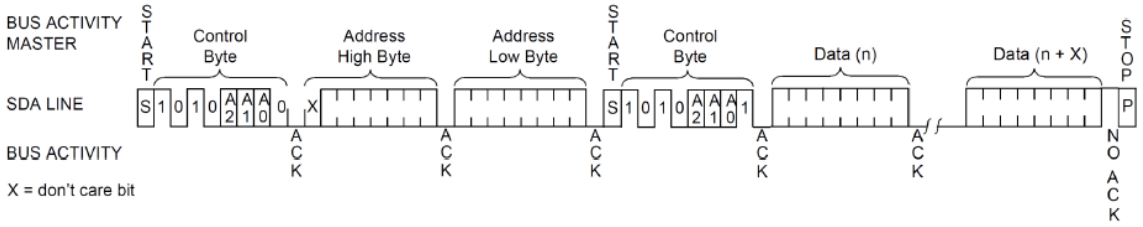
9.1.3. Lectura de un byte

La primera parte de la operación de lectura es igual a una operación de escritura (bit de START más byte de control más bit en '0', seguido de la dirección a acceder), pero luego se envía un nuevo bit de START seguido del byte de control más un bit en '1'. A continuación, la memoria transmitirá el byte de dato.



9.1.4. Lectura secuencial

La EEPROM retiene en un contador interno la última dirección accedida (tanto para lectura como para escritura), incrementada en 1. Esto no sólo permite un acceso inmediato al dato apuntado, sino también una lectura secuencial de bytes consecutivos en la memoria (sin un límite como en la escritura de 64 bytes, sino que se puede leer la memoria entera).



9.2. Ejemplo de aplicación: escritura/lectura de sensores inerciales

9.2.1. Sensor MPU6050 con giróscopo y acelerómetro

Este sensor requiere una conexión de dos líneas (**SDA** y **SCL**), soporta modos de transferencia estándar (**100kHz**) y rápido (**400kHz**) y permite lecturas y escrituras tanto simples como múltiples. La dirección del esclavo viene dada por los 7 bits: $\mathbf{1\ 1\ 0\ 1\ 0\ 0}\ \boldsymbol{AD0}$, siendo ***AD0*** un pin físico accesible del mismo. Es decir que podemos acceder a 2 sensores MPU6050 como máximo desde el maestro, uno con su pin ***AD0*** conectado a tierra y el otro con su pin ***AD0*** conectado a $\boldsymbol{V_{dd}}$.

9.2.2. Sensor ADXL345 con acelerómetro

Al igual que el MPU6050, este sensor requiere una conexión de dos líneas (**SDA** y **SCL**), soporta modos de transferencia estándar (**100kHz**) y rápido (**400kHz**) y permite lecturas y escrituras tanto simples como múltiples. En particular, la interfaz $\boldsymbol{I^2C}$ se activa sólo si el pin físico $\overline{\boldsymbol{CS}}$ del sensor se conecta a $\boldsymbol{V_{dd}}$. Ejemplo: con el pin físico $\boldsymbol{SDO/ALT\ ADDRESS}$ del sensor conectado a $\boldsymbol{V_{dd}}$, la **dirección** para acceder a este dispositivo es $\boldsymbol{0x1D}$, seguido del bit $\boldsymbol{R/W}$, lo que se traduce en una dirección $\boldsymbol{0x3A}$ para **escritura**, y $\boldsymbol{0x3B}$ para **lectura**.

9.3. Interfaz i2c en Atmega328p

En los microcontroladores *AVR* esta interfaz se denomina TWI (two-wire interface), por cuestiones de patentes. Presenta las siguientes funcionalidades básicas:

- Maestro o esclavo
- Direccionamiento de 7 bits
- Soporta arbitraje multi-master
- Velocidad hasta 400kbps
- Soporta interrupciones

Para comenzar, debe ponerse en '0' el bit PRTWI del registro de reducción de energía, para que se habilite la interfaz TWI. El primer registro asociado es el ***TWBR***, el cual establece la frecuencia de oscilación de SCL (bit rate) de la comunicación, para modo maestro.

Bit	7	6	5	4	3	2	1	0
(0xB8)	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
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

El bit rate se calcula como:

$$F_{SCL} = \frac{F_{CPU}}{16 + 2.(TWBR).(prescaler\ value)}$$

Por lo que, para una F_{SCL} deseada, se deberá escribir en este registro:

$$TWBR = \frac{\frac{F_{CPU}}{F_{SCL}} - 16}{2.(prescaler\ value)}$$

El próximo registro asociado es el ***TWCR*** (registro de control).

Bit	7	6	5	4	3	2	1	0
(0xBC)	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W
Initial Value	0	0	0	0	0	0	0	0

- ***TWINT***: flag de interrupción. Se setea cuando la comunicación ha terminado una tarea, y no se vuelve automáticamente a 0. Mientras este bit está seteado, se estira el período de estado bajo de SCL. Todos los accesos al registro de datos, de estado o de dirección deben hacerse antes de apagar este bit, ya que al apagarlo comienza su próxima tarea.
- ***TWEA***: bit de habilitación del pulso de "acknowledge"
- ***TWSTA***: condición de START
- ***TWSTO***: condición de STOP
- ***TWWC***: detector de colisión
- ***TWEN***: habilitación de comunicación TWI
- ***TWIE***: habilitación de interrupción de TWI

El próximo registro asociado es el ***TWSR*** (registro de estado).

Bit	7	6	5	4	3	2	1	0
(0xB9)	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0
Read/Write	R	R	R	R	R	R	R/W	R/W
Initial Value	1	1	1	1	1	0	0	0

Los bits ***TWS7:3*** reflejan el **estado** de los dos hilos del bus y la lógica de la comunicación, según la siguiente tabla:

Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCRn				
			STA	STO	TWINT	TWEA	
0x08	A START condition has been transmitted	Load SLA+W	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
0x10	A repeated START condition has been transmitted	Load SLA+W or	0	0	1	X	SLA+W will be transmitted; ACK or NOT ACK will be received
		Load SLA+R	0	0	1	X	SLA+R will be transmitted; Logic will switch to Master Receiver mode
0x18	SLA+W has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x20	SLA+W has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x28	Data byte has been transmitted; ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	1	0	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset

Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware	Application Software Response					Next Action Taken by TWI Hardware
		To/from TWDR	To TWCRn				
			STA	STO	TWINT	TWEA	
0x30	Data byte has been transmitted; NOT ACK has been received	Load data byte or	0	0	1	X	Data byte will be transmitted and ACK or NOT ACK will be received
		No TWDR action or	1	0	1	X	Repeated START will be transmitted
		No TWDR action or	0	1	1	X	STOP condition will be transmitted and TWSTO Flag will be reset
		No TWDR action	1	1	1	X	STOP condition followed by a START condition will be transmitted and TWSTO Flag will be reset
0x38	Arbitration lost in SLA+W or data bytes	No TWDR action or	0	0	1	X	2-wire Serial Bus will be released and not addressed Slave mode entered
		No TWDR action	1	0	1	X	A START condition will be transmitted when the bus becomes free

Los bits ***TWPS1:0*** de lectura/escritura controlan el prescaler del "bit rate".

TWPS1:0	prescalerValue
00	1
01	4
10	16
11	64

Para modo esclavo, se usan los registros ***TWAR*** y ***TWAMR***.

Bit	7	6	5	4	3	2	1	0	
(0xBA)	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	TWAR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	1	1	1	1	1	1	0	
Bit	7	6	5	4	3	2	1	0	
(0xBD)	TWAM[6:0]							–	TWAMR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	
Initial Value	0	0	0	0	0	0	0	0	

En los 7 bits más significativos del registro ***TWAR*** se debe guardar la dirección de esclavo del micro para que éste sea accedido por el maestro de la comunicación. El registro ***TWAMR*** sirve de máscara para la dirección de esclavo.

Por último, el registro de datos ***TWDR*** es el que contiene el próximo dato a ser transmitido o el último dato recibido, según el modo de operación. Éste mantiene estable su dato mientras esté en proceso el desplazamiento de un byte (tiempo durante el cual el flag de interrupción ***TWINT*** se setea por hardware). Además, no se lo puede escribir cuando dicho flag está seteado.

10. Módulo conversor A/D

Este módulo del microcontrolador permite la *conversión* de un voltaje en un cierto rango analógico (por ejemplo, de 0 a 5V) en una representación binaria proporcional, con una determinada resolución (normalmente entre 8 y 24 bits). Las arquitecturas electrónicas para llevar a cabo esta conversión son muy variadas. Las más rápidas (GigaS/s) y de baja resolución (8 bits), son flash ADC y pipeline ADC. Las más lentas (kS/s) y de alta resolución (24 bits) son sigma-delta y doble rampa. En los microcontroladores se utilizan conversores A/D que trabajan por aproximaciones sucesivas, con una resolución de 10 a 12 bits. Hay algunos especializados con conversores sigma-delta de hasta 24 bits de resolución.

Los microcontroladores disponen de varios ***canales analógicos*** que son seleccionados mediante un multiplexor interno, para su conversión en un único núcleo A/D. Las tasas de muestreo sobre un mismo canal varían desde los 50kS/s en los micros de gama baja-media, hasta unos 7MS/s en los de gama alta.

10.1. Conversor A/D en PIC18F2550

Este módulo tiene **10 entradas**, y permite la conversión de una señal de entrada analógica a un número digital correspondiente de **10 bits**. Se maneja con los siguientes registros:

Registros ***ADRESH*** y ***ADRESL***, con el resultado alto y bajo de la conversión A/D, respectivamente.

Registro ***ADCON0***: controla la ***OPERACIÓN*** del módulo A/D.

REGISTER 21-1: ADCON0: A/D CONTROL REGISTER 0

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

- **CHS3:0** → selección del **Canal Digital**.

CHS3:0	Canal Analógico
0000	Canal 0 (AN0)
0001	Canal 1 (AN1)
0010	Canal 2 (AN2)
0011	Canal 3 (AN3)
0100	Canal 4 (AN4)
1000	Canal 8 (AN8)
1001	Canal 9 (AN9)
1010	Canal 10 (AN10)
1011	Canal 11 (AN11)
1100	Canal 12 (AN12)

Ejemplo: al escribir 0bXX0000XX en este registro, seleccionamos el Canal 0 (AN0).

- bit **1**: bit de **Estado de conversión** A/D (cuando **ADON**=1) → 1: conversión en progreso, 0: estado ocioso.
- bit **0**: bit de **Habilitación del módulo** A/D → 1: habilitado, 0: no habilitado.

Registro ***ADCON1***: configura las ***FUNCIONES DE LOS PINES*** del puerto.

REGISTER 21-2: ADCON1: A/D CONTROL REGISTER 1

U-0	U-0	R/W-0	R/W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG0	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

- bit **5**: bit de Configuración del **Voltaje de Referencia (-)** → 1: AN2, 0 Vss
- bit **4**: bit de Configuración del **Voltaje de Referencia (+)** → 1: AN3, 0 Vdd
- bits **3-0**: bits de Configuración del **Puerto A/D**

PCFG3:PCFG0	AN12	AN11	AN10	AN9	AN8	AN4	AN3	AN2	AN1	AN0
0000	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A
0110	D	D	D	D	A	A	A	A	A	A
0111	D	D	D	D	D	A	A	A	A	A
1000	D	D	D	D	D	A	A	A	A	A
1001	D	D	D	D	D	A	A	A	A	A
1010	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D

Registro ***ADCON2***: configura la ***FUENTE DE CLOCK***, ***TIEMPO DE ADQUISICIÓN*** y ***JUSTIFICACIÓN***.

REGISTER 21-3: ADCON2: A/D CONTROL REGISTER 2

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7							bit 0

- bit **7**: bit de selección del **Formato del Resultado** A/D (1: justificado a derecha, 0: justificado a izquierda.
- bit **5-3**: bits de selección del **Tiempo de Adquisición** A/D

ACQT2:ACQT0	Canal Analógico
111	20 T_{AD}
110	16 T_{AD}
101	12 T_{AD}
100	8 T_{AD}
011	6 T_{AD}
010	4 T_{AD}
001	2 T_{AD}
000	0 T_{AD}

- bit **2-0**: bits de selección de la **Fuente de Clock** de conversión A/D

ADCS2:ADCS0	Canal Analógico
111	F_{RC}
110	$F_{OSC}/64$
101	$F_{OSC}/16$
100	$F_{OSC}/4$
011	F_{RC}
010	$F_{OSC}/32$
001	$F_{OSC}/8$
000	$F_{OSC}/2$

Cuando una conversión A/D se completa, el resultado se guarda en el par de registros ADRESH:ADRESL, el bit GO/DONE se apaga y el bit ADIF se enciende.

10.2. Conversor A/D en Atmega328p

Este módulo tiene **6 canales** de entradas simples multiplexadas, permitiendo la conversión de señales analógicas (en un rango de 0 a Vcc) a un número digital con una resolución de **10 bits**. Dicha conversión se realiza mediante aproximaciones sucesivas, con una velocidad máxima de 76.9kSps, o hasta 15kSps a máxima resolución. El mínimo valor representa $\boldsymbol{GND}$ y el máximo se puede elegir como un voltaje de referencia de 1.1V, una fuente externa en $\boldsymbol{AREf}$ o puentear $\boldsymbol{AREf}$ con el pin $\boldsymbol{AVcc}$ (colocando un capacitor para filtrar ruidos eléctricos).

El conversor A/D cuenta además con un circuito de "Sample and Hold" que congela el valor de la entrada durante cada conversión. Permite interrupción por conversión completa A/D. La conexión con el multiplexor analógico permite 6 entradas de voltaje simples a través de los pines 0 a 5 del puerto C, todas con su referencia negativa a GND.

En la siguiente tabla se ven los ciclos de reloj de AD que toman las distintas operaciones:

Table 28-1. ADC Conversion Time

Condition	Sample & Hold (Cycles from Start of Conversion)	Conversion Time (Cycles)
First conversion	13.5	25
Normal conversions, single ended	1.5	13
Auto Triggered conversions	2	13.5

Para empezar, el bit ***PRADC*** del registro de reducción de energía ***PRR*** debe estar apagado para que se pueda habilitar este módulo.

El siguiente registro asociado a este módulo es el ***ADMUX***.

Bit	7	6	5	4	3	2	1	0
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
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

- ***MUX3:0*** → selección de canal de entrada analógico

MUX3:0	Canal
0 a 7	Canal
8	Sensor T°
14	1.1 (Vref)
15	GND

- ***ADLAR*** → alineación de datos (1: izquierda, 0: derecha)
- ***REFS1:0*** → selección de tensión de referencia

TWPS1:0	Tensión de referencia
00	ARef
01	AVcc
11	Interna 1.1

Nota: **AVcc** no debe diferir en más de 0.3V de Vcc.

El siguiente registro asociado es el ***ADCSRA***.

Bit	7	6	5	4	3	2	1	0
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
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

- ***ADEN*** → habilitación del conversor (1: habilita, 0: deshabilita)
- ***ADSC*** → inicia la conversión (permanecerá en alto durante la conversión, y se reiniciará por hardware cuando ésta se complete).
- ***ADATE*** → habilitación de auto-trigger. Si está habilitado, un flanco positivo en la señal seleccionada (ver próximo registro), resetea el prescaler del ADC y comienza una conversión.
- ***ADIF*** → flag de interrupción por fin de conversión
- ***ADIE*** → habilitación de interrupción por fin de conversión
- ***ADPS2:0*** → selección del prescaler del conversor A/D

ADPS2:0	prescalerValue
000 ó 001	2
010	4
011	8
100	16
101	32
110	64
111	128

El siguiente registro asociado es el ***ADCSRB***.

Bit	7	6	5	4	3	2	1	0
(0x7B)	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0
Read/Write	R	R/W	R	R	R	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

- ***ADTS2:0*** → selección del auto-trigger

ADTS[2:0]	Trigger Source
000	Free Running mode
001	Analog Comparator
010	External Interrupt Request 0
011	Timer/Counter0 Compare Match A
100	Timer/Counter0 Overflow
101	Timer/Counter1 Compare Match B
110	Timer/Counter1 Overflow
111	Timer/Counter1 Capture Event

- ***ACME*** → para modo comparador

El registro ***DIDR0*** permite deshabilitar los buffers de entrada digital:

Bit	7	6	5	4	3	2	1	0	
(0x7E)	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	DIDR0
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Cuando estas entradas son analógicas, se escribe un '1' en su bit correspondiente para deshabilitar su buffer de entrada digital, a modo de reducir el consumo de energía.

Por último, los registros de datos ***ADCL*** y ***ADCH***, para los dos tipos de alineación:

Con ADLAR=0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL

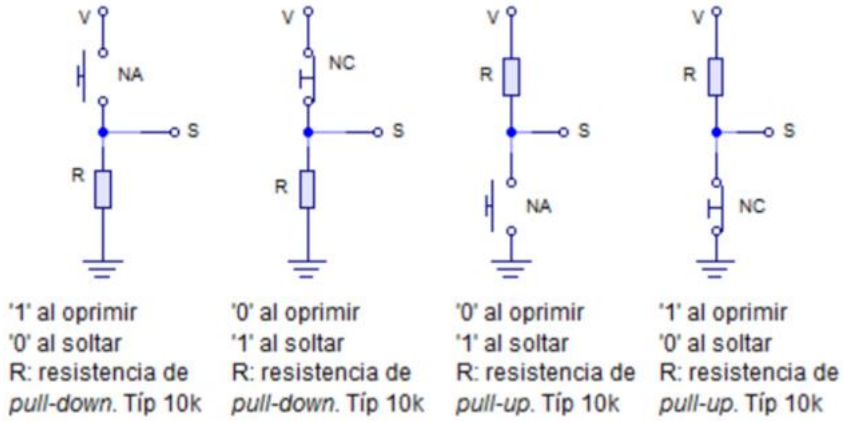
Con ADLAR=1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL

11. Consideraciones prácticas

11.1. Pull-up, pull-down

Si se utilizan pines como entradas digitales al microcontrolador, por ejemplo para detectar un cambio de estado a través un pulsador, se debe asegurar que en todo momento los mismos tengan un valor totalmente definido. Si algún pin queda sin referencia (GND o Vdd), su estado es indeterminado y puede presentar un comportamiento errático. Por lo tanto, se debe utilizar alguna de las siguientes conexiones, según el estado o tipo de flanco que se desee detectar:



'1' al oprimir '0' al soltar R: resistencia de *pull-down*. Típ 10k	'0' al oprimir '1' al soltar R: resistencia de *pull-down*. Típ 10k	'0' al oprimir '1' al soltar R: resistencia de *pull-up*. Típ 10k	'1' al oprimir '0' al soltar R: resistencia de *pull-up*. Típ 10k

Además, el contacto mecánico del pulsador no es perfecto, por lo que al pulsarlo y soltarlo se introduce "ruido" a través del pin. Es por esto que, si no se toman las medidas necesarias, podrían detectarse múltiples flancos ante una sola pulsada, perdiendo determinismo en el comportamiento del sistema.

Nota: Agregar un condensador de 0.1uF del punto S (entrada al microcontrolador) a GND ayuda a filtrar los ruidos eléctricos de alta frecuencia. Esto se suele complementar con estrategias de software que ayudan a mejorar la respuesta, como vemos en los ejemplos.

Ver consideraciones prácticas: "Capacitores de desacoplo" [11.2].

11.2. Capacitores de desacoplo

Estos capacitores se utilizan para eliminar ruidos externos de alta frecuencia, provenientes ya sea de cargas inductivas, armónicos generados por conmutación, "rebote" mecánico en pulsadores, interferencias electromagnéticas, etc. Se colocan entre el terminal que se quiere proteger (deben colocarse lo más cerca posible del mismo) y la tierra del circuito. Valores típicos: $100nF$, condensadores cerámicos o de tantalio.

La impedancia de una carga capacitiva está dada por:

$$X_C = \frac{1}{2\,\pi\,f\,C}$$

Al ser inversamente proporcional a la frecuencia, ante una tensión continua y limpia ($f = 0$) es como si estos capacitores no estuvieran presentes en el circuito (impedancia infinita). Por el contrario, para frecuencias altas la impedancia es baja, y los capacitores se comportan como un corto-circuito a tierra. Es decir, esas señales no entran al terminal protegido, sino que se van a tierra.

De esta manera, colocando capacitores en los pines de alimentación de cada chip del circuito, entradas y salidas de reguladores de tensión, pulsos provenientes de un interruptor, etc., se obtiene un sistema más robusto.

Adicionalmente, se colocan **capacitores electrolíticos**, del orden de 10 a $100\mu F$, en paralelo con los capacitores de desacoplo.

Cuando están presentes cargas inductivas que se alimentan de la misma fuente o derivada de ella, se presentan consumos instantáneos de alta corriente (tiempos del orden de ms). Si la fuente no puede suministrar dicha corriente, estos capacitores apoyan en esos instantes dando su carga, y así no se cae el voltaje (que puede por ejemplo resetear un microcontrolador). A mayor capacidad, más seguridad en casos críticos.

11.3. Dirección de pines por defecto

Por lo general, todos los pines de propósito general son entradas al energizar el microcontrolador, por lo que configurarlos de esa forma (ej. TRISx = 1) puede ser redundante. Por el contrario, si se desea utilizar pines como *salidas*, sí se los debe configurar de esa forma (ej. TRISx = 0) y luego asignarles un estado inicial, por ejemplo apagado (ej. PORTx = 0).

Esto se puede verificar observando los valores por defecto de los registros de direccionamiento. Por ejemplo, en el PIC16F628A, para sus puertos A y B:

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR Reset(1)
85h	TRISA	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	1111 1111
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111

Lo mismo ocurre para el Atmega328p con sus puertos B, C y D, y para el Atmega2560 con sus puertos A, B, C, D, E, F, G, H, J, K y L. Tener en cuenta que para los *AVR*, un 0 en el registro de direccionamiento (DDRx) es entrada, contrario a los *PIC*.

Sin embargo, no todos los pines traen el mismo valor por defecto en sus registros de estado. Por ejemplo, en el PIC16F628A, algunos pines vienen en estado bajo (0) y otros en estado indeterminado (x) en sus puertos A y B.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR Reset(1)
05h	PORTA	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0	xxxx 0000
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx

Por motivos de seguridad, antes de declarar como salida a cualquier pin, se pone en 0 su bit de estado. Si dicho pin activara por ejemplo un motor, y si éste tuviera un estado alto al momento de declararlo como salida, se activaría el motor durante la inicialización. Debe tenerse en cuenta que, en los micros *AVR*, si se asigna un estado alto a un pin que es entrada, automáticamente se activa una resistencia de pull-up interna (de $100k\Omega$) en ese pin.

Por otro lado, para evitar ruidos eléctricos, todos los pines que no se utilicen del microcontrolador, se suelen declarar como salidas en estado bajo y luego se los conecta a GND en el circuito físico.

Nota: en Proteus los cuadrados rojos en los pines indican un estado alto de tensión, los azules un estado bajo y los grises un estado indeterminado o desconectado. Esto sirve para verificar rápidamente la correcta inicialización de todos los pines al momento de energizar el microcontrolador.

11.4. Error de baudrate

En la comunicación UART existe un máximo error admisible de velocidad de envío de caracteres, entre emisor y receptor. Además, si uno de ellos tiene un error por defecto y el otro por exceso, aumenta el desfasaje de velocidades. En los catálogos de los microcontroladores encontramos, en función del cristal utilizado, el error de baudrate que se comete respecto de cada velocidad normalizada. En los *AVR*, utilizando un cristal de 16MHz, y según si trabaja con modo simple o doble velocidad (con el bit U2Xn en 0 o en 1, respectivamente) se tiene:

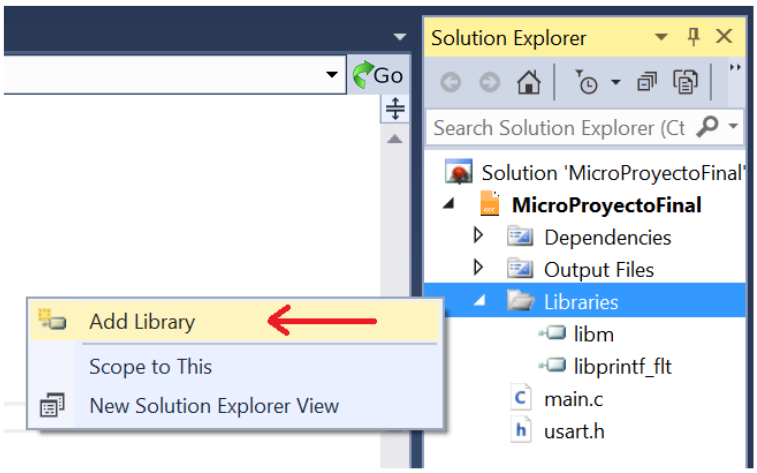
Baud Rate [bps]	f_{osc} = 16.0000MHz			
	U2Xn = 0		U2Xn = 1	
	UBRRn	Error	UBRRn	Error
2400	416	-0.1%	832	0.0%
4800	207	0.2%	416	-0.1%
9600	103	0.2%	207	0.2%
14.4k	68	0.6%	138	-0.1%
19.2k	51	0.2%	103	0.2%
28.8k	34	-0.8%	68	0.6%
38.4k	25	0.2%	51	0.2%
57.6k	16	2.1%	34	-0.8%
76.8k	12	0.2%	25	0.2%
115.2k	8	-3.5%	16	2.1%
230.4k	3	8.5%	8	-3.5%
250k	3	0.0%	7	0.0%
0.5M	1	0.0%	3	0.0%
1M	0	0.0%	1	0.0%
Max.(1)	1Mbps		2Mbps	

Por ejemplo, si se desea trabajar a 115200bps con la USART0, se comete menos error con el UBRR0 correspondiente a 57600bps (16), en modo doble velocidad (U2X0=1). Por otro lado, si se programa a dos micros *AVR* (que trabajan con el mismo cristal) para que se comuniquen por la USART0 entre ellos, y se escribe el mismo valor en sus registros UBRR0, el error de baudrate será nulo (incluso se puede elegir una velocidad no normalizada).

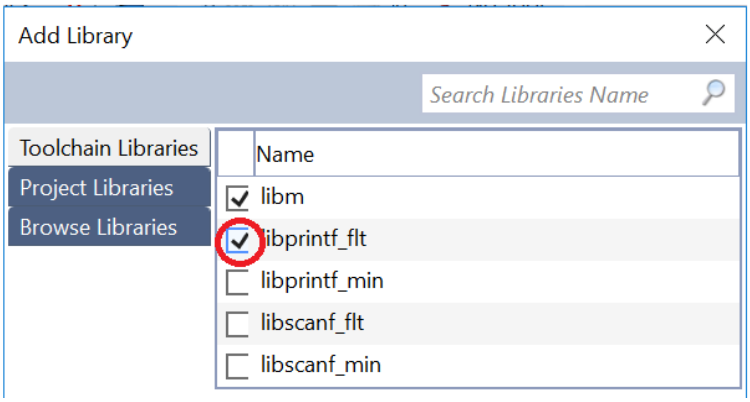
11.5. Números decimales por puerto serie en AVR

Para poder imprimir números de punto flotante por la UART, se deben realizar los siguientes pasos adicionales. Una vez creado el proyecto con AtmelStudio:

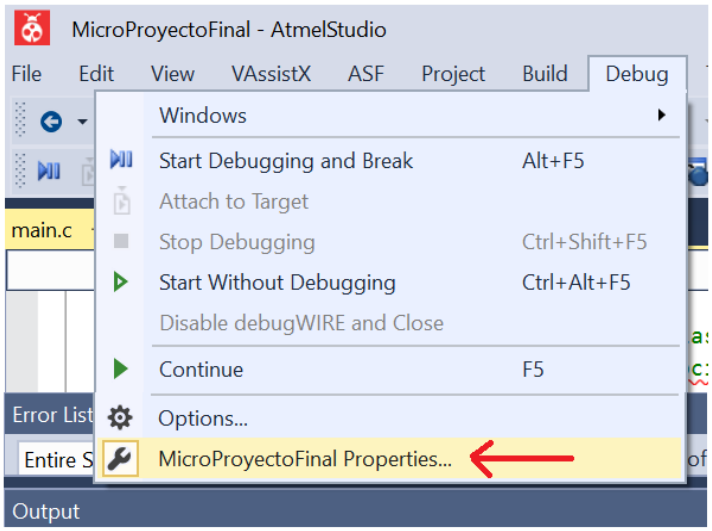
1) Agregar librería, haciendo click derecho en Libraries dentro de la carpeta solución del proyecto y seleccionando Add Library.



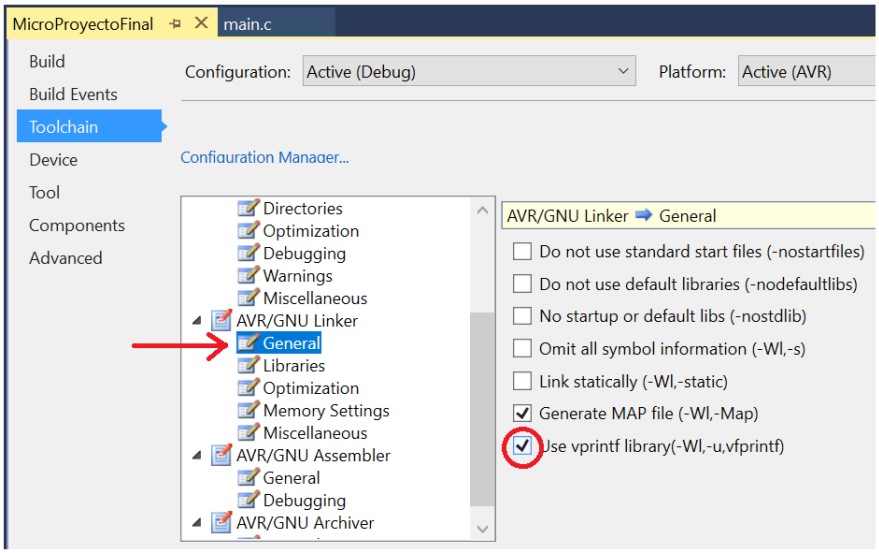
2) Tildar la pestaña "libprintf_flt".



3) Ir a las propiedades del proyecto en *"Debug"* → *"Project Properties"*.



4) Tildar la pestaña *"Use vprintf library"* en *"Toolchain"* → *"AVR/GNU Linker"* → *"General"*.



Nota: para los usuarios de Mac o Linux, esto mismo se logra accediendo al makefile del proyecto, comentando la línea que incluye *libprintf* y descomentando la línea que incluye *libprintf_flt*.

11.6. Overhead

El *overhead* es el tiempo adicional que se tarda en ir y volver a la rutina de la interrupción de un TIMER. Una forma de medir este tiempo es simulando en Proteus, introduciendo en el *PIC* virtual el archivo con extensión "*.cof*" generado al compilar, para poder ejecutar en modo depuración. Luego corremos la simulación y la pausamos. Aparecerá una ventana con el código de programa, donde podemos introducir un *breakpoint* dentro del cuerpo de la interrupción (doble click en la línea de código), como se ve en la siguiente imagen.

```
--------  //*************************************************************************//
--------  //                  Interrupción por Desborde de TIMER 1                   //
032C      //*************************************************************************//
--------  #int_TIMER1  // DURACIÓN LECTURAS
--------  void  TIMER1_isr( void) // entra cada 0.04s
--------  {
0420            set_timer1( 0xFFFF +OVERHEAD -PULSOS_RESTA);
0428            cont_TMR1 ++;
042E            output_toggle(PIN_C2); // PARPADEA LED VERDE (cada 0.04s siempre)
0432            if(cont_TMR1  == time_TMR1)  // default: entra cada 4s
--------        {
043E                  ON         = 0;
```

Al correr la simulación de forma cíclica se observará, en la barra de la parte inferior de la pantalla, el tiempo que tarda en ir y volver al mismo punto marcado, en distintas ejecuciones.

Nota: se debe ejecutar más de una vez, ya que a la primera contará el tiempo desde que pausamos la simulación (que no necesariamente coincide con el breakpoint).

La diferencia entre este tiempo medido y el tiempo esperado es el *overhead*, el cual hay que dividirlo por el tiempo de ciclo para obtener el número de ciclos que lo generan. Además, se debe tener en cuenta el postscaler utilizado. En este caso:

- tiempo esperado: 40ms
- tiempo medido: 40.02ms
- diferencia: 0.02ms
- período de ciclo: 0.2us
- número de ciclos que generan esta diferencia: 0.02ms/0.2us = 100 ciclos
- postscaler en 4 → 100/4=25 ciclos → OVERHEAD = **25**

Con el overhead definido con su valor adecuado (en la imagen anterior éste estaba definido como 0), si se repite el procedimiento anterior, se podrá verificar que el tiempo entre entradas consecutivas a la rutina de la interrupción es el deseado:

11.7. Selección de prescaler, base de tiempo

Al generar tiempos en microcontroladores se debe tener en cuenta la frecuencia de trabajo (T_{ciclo}) del mismo, que es por ejemplo igual a la frecuencia de clock en los *AVR* o un cuarto de la misma en los *PIC*. Se debe tener en cuenta además que un timer de N bits desborda al transcurrir 2^N ciclos, por lo que el tiempo máximo que se puede generar entre desbordes, con prescaler en 1, será:

$$t_{MAX} = T_{ciclo} * 2^N.$$

$$t_{MAX} = \frac{1}{F_{ciclo}} * 2^N.$$

Si se desea generar un tiempo mayor al anterior, se deberá seleccionar el prescaler adecuado entre los disponibles (Ej.: 1, 8, 64, 128, 256 o 1024), con lo cual se obtendrá:

$$t_{MAX_{prescaler}} = prescaler * \frac{1}{F_{ciclo}} * 2^N.$$

Si el tiempo que se desea generar es aún mayor que el obtenido con el máximo prescaler, una solución es generar una base de tiempo fija e implementar un contador de dicha base. Por ejemplo, con una $F_{ciclo} = 16MHz$ en un micro *AVR*, con un timer de 16 bits y un prescaler máximo de 1024, el máximo tiempo entre desbordes es:

$$t_{MAX_{1024}} = 1024 * \frac{1}{16 * 10^6} * 2^{16} = 4.194303seg.$$

Si se desea realizar una tarea cada 10 segundos, se puede elegir una base de tiempo de 1 segundo, obtenida por ejemplo con prescaler en 256 y tope en 62500:

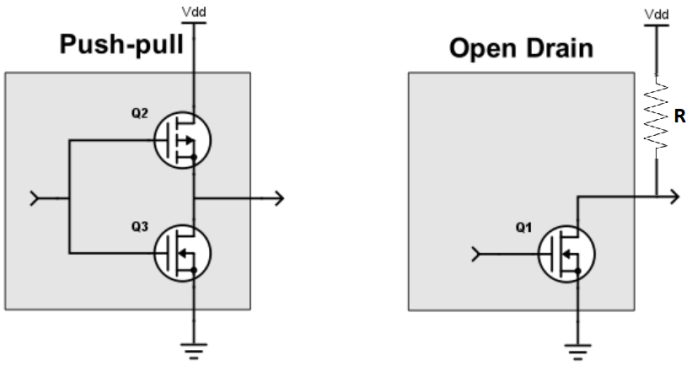
$$t_{base} = 256 * \frac{1}{16 * 10^6} * 62500 = 1\ seg.$$

Luego, en la rutina de interrupción asociada, se puede incrementar un contador (inicializado en 0) en una unidad cada vez que se entra a la misma. Cuando el contador alcanza el valor 10, se realiza la tarea deseada y se reinicia el mismo a cero.

11.8. Salida complementaria vs colector abierto

La salida ***complementaria*** (push-pull) usa dos transistores (superior e inferior), los cuales estarán activos complementariamente (uno sí y el otro no) para poner a la salida el nivel apropiado de tensión. La activación del transistor superior da un nivel alto y la activación del transistor inferior da un nivel bajo.

La salida ***colector abierto*** (open-drain) tiene sólo un transistor inferior y una resistencia de pull-up. Por lo tanto, activando y desactivando el transistor se lograr un nivel bajo y alto de tensión a la salida, respectivamente.



Los microcontroladores en general traen pines con ambos tipos de salida, y/o permiten establecerlas escribiendo en bits de registros que activan/desactivan el transistor superior y la resistencia de pull-up interna (si la tiene).

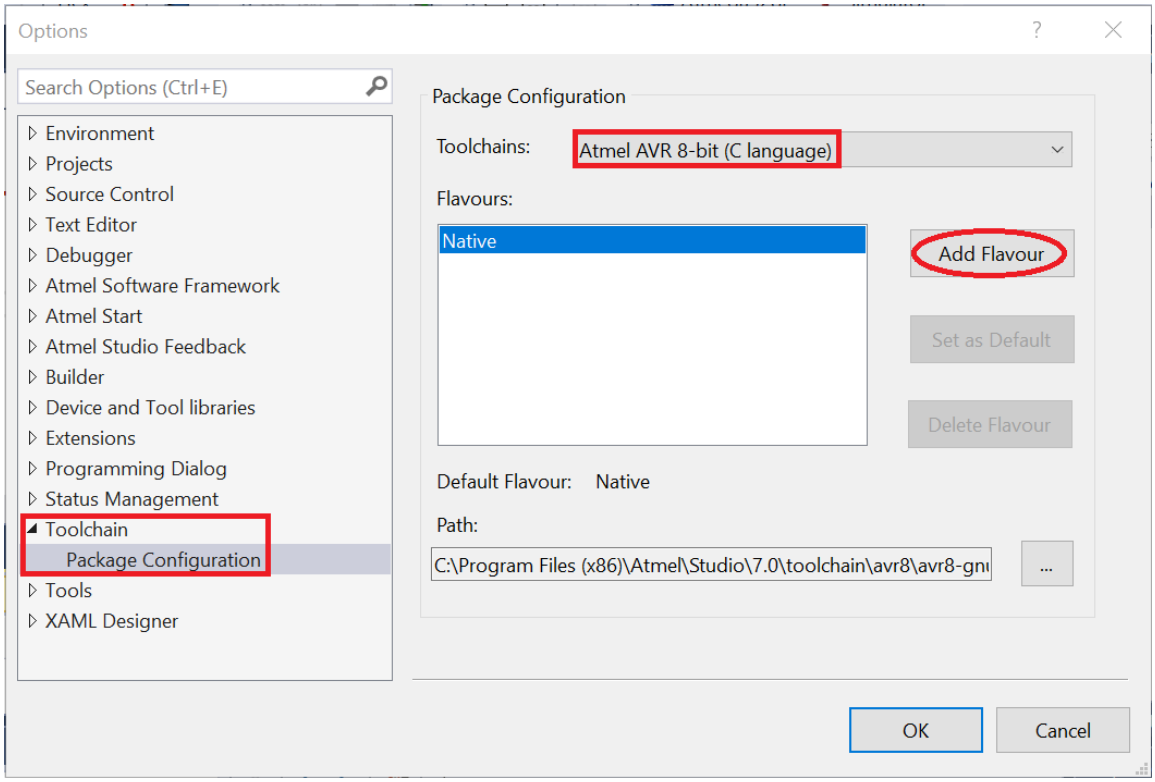
Una ventaja que brinda la salida complementaria es mayor velocidad, porque la línea es conducida de forma instantánea en ambos sentidos. En cambio, en colector abierto, la línea sólo puede conducirse de forma instantánea al nivel bajo. El tiempo de conducción al nivel alto depende de la constante RC (resistencia de pull-up y capacitancia parásita del pin y de la placa). Una forma de aumentar la velocidad al nivel alto es disminuir el valor de la resistencia, sin embargo, al disminuir ésta aumenta el consumo de corriente al establecer el nivel bajo, la cual tiene que drenar el transistor. Por lo tanto, hay un compromiso entre velocidad y consumo.

Por otro lado, la ventaja del colector abierto es que permite el uso de múltiples salidas en la misma línea, en una conexión llamada "AND cableada" por su comportamiento. Por ejemplo, se pueden conectar varios pines de colector abierto de un microcontrolador a una misma salida (o muchos transmisores colector abierto a un mismo bus), con una única resistencia de pull-up. Con cualquier pin en nivel bajo, habrá un nivel bajo a la salida. Sólo con todos los pines en nivel alto, habrá un nivel alto a la salida. Es decir, si dos o más pines intentan transmitir al mismo tiempo, darán una salida baja si conducen distinto (uno alto y otro bajo), sin cortocircuito. En cambio, en salida complementaria si dos o más pines conducen distinto al mismo tiempo, se produce cortocircuito. Con este tipo de salidas se debe asegurar la habilitación de los transmisores de manera excluyente. También se suelen usar puertas de tercer estado.

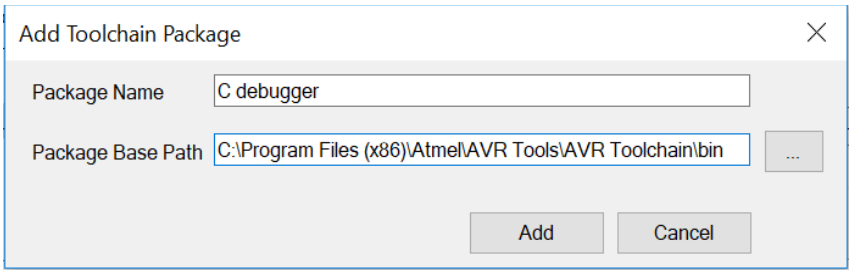
11.9. Depuración en AVR

Por defecto, en las versiones actuales AtmelStudio no se permite la depuración desde en un programa externo. Para esto instalamos el archivo de la carpeta "AVR toolchain" dentro de MyEP2020 [1].

Luego vamos a *Tools* → *Options* → *Toolchain* → *Package Configuration*. Seleccionamos *Atmel AVR 8-bit (C language)* → *Add Flavour*.

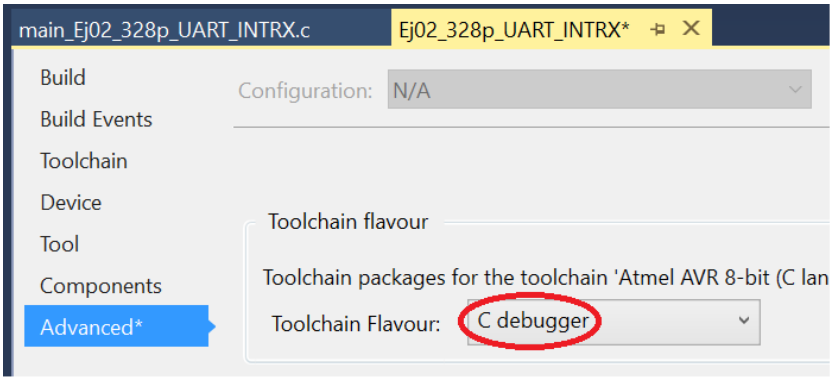


Elegimos un nombre (ejemplo C debugger) y damos el directorio de la carpeta bin del toolchain instalado en el paso anterior.

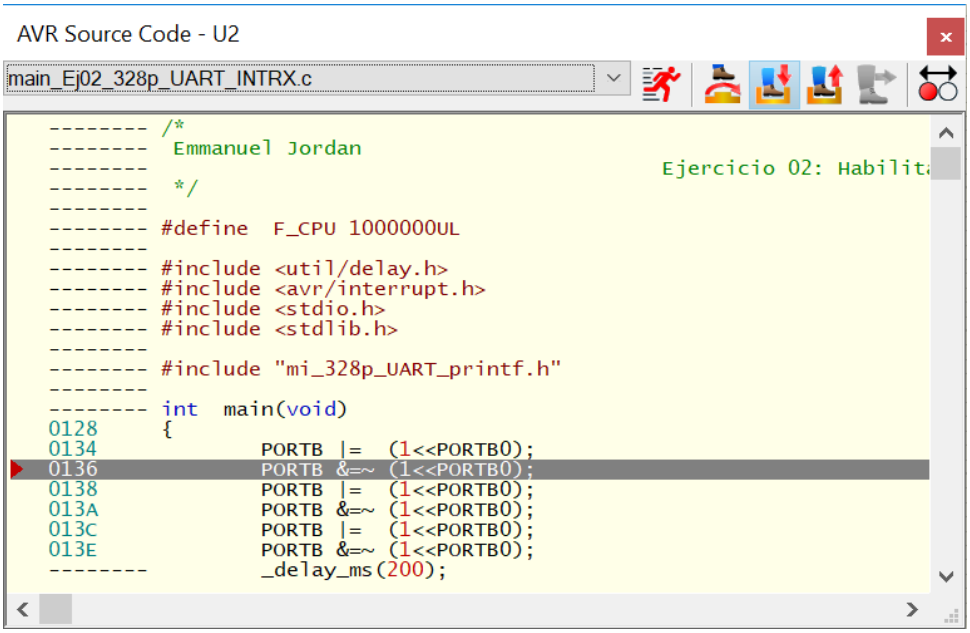


En la ventana anterior seleccionamos la nueva pestaña (*C debugger*) y elegimos *Set As Default* y *OK*. Click derecho a proyecto → *Clean*. Click derecho a solución → *Clean Solution*.

Proyecto → *Properties* → *Advanced* → *Toolchain Flavour: C debugger.*



Hecho esto, ya podemos compilar nuestro proyecto e ingresar el correspondiente archivo .elf por ejemplo en Proteus, para ejecutar el mismo en modo depuración.



Bibliografía

[1] J. E, «MyEP2020,» [En línea]. Available: https://drive.google.com/drive/folders/0B-0P_4m3NjGYNWl6aGIsdHEtZUk.