Unidad 3:

Microcontroladores y otros sistemas embebidos. Subsistemas de comunicación y de control de movimiento

¿Cómo elegir un sistema embebido?

Conectividad:

Trazar el esquema tecnológico, con todas las interfaces entre el controlador y el entorno.

- Pines de E/S general.
- E/S especiales:
    - Canales A/D. Salidas PWM. Entradas QEI
- Timers
- Comunicaciones I2C, SPI (se pueden emular por software, con carga al µC)
- Comunicaciones UART, CAN, Ethernet, USB.

Algunos subsistemas (Ej A/D) pueden ser externos, conectados por I2C o SPI

Capacidad de procesamiento, memoria, velocidad de respuesta a eventos:

Estimar cantidad y tamaño de variables → **RAM**

Especificar restricciones temporales (eventos críticos) y algoritmos más costosos computacionalmente.

- Velocidad (MIPS, DMIPS).
- Si tiene multiplicación en 1 ciclo, módulo DSP, módulo de Punto Flotante.
- Latencia en interrupciones/vectorización/anidamiento

Considerar la complejidad del programa (en cantidad de rutinas distintas) → **ROM**

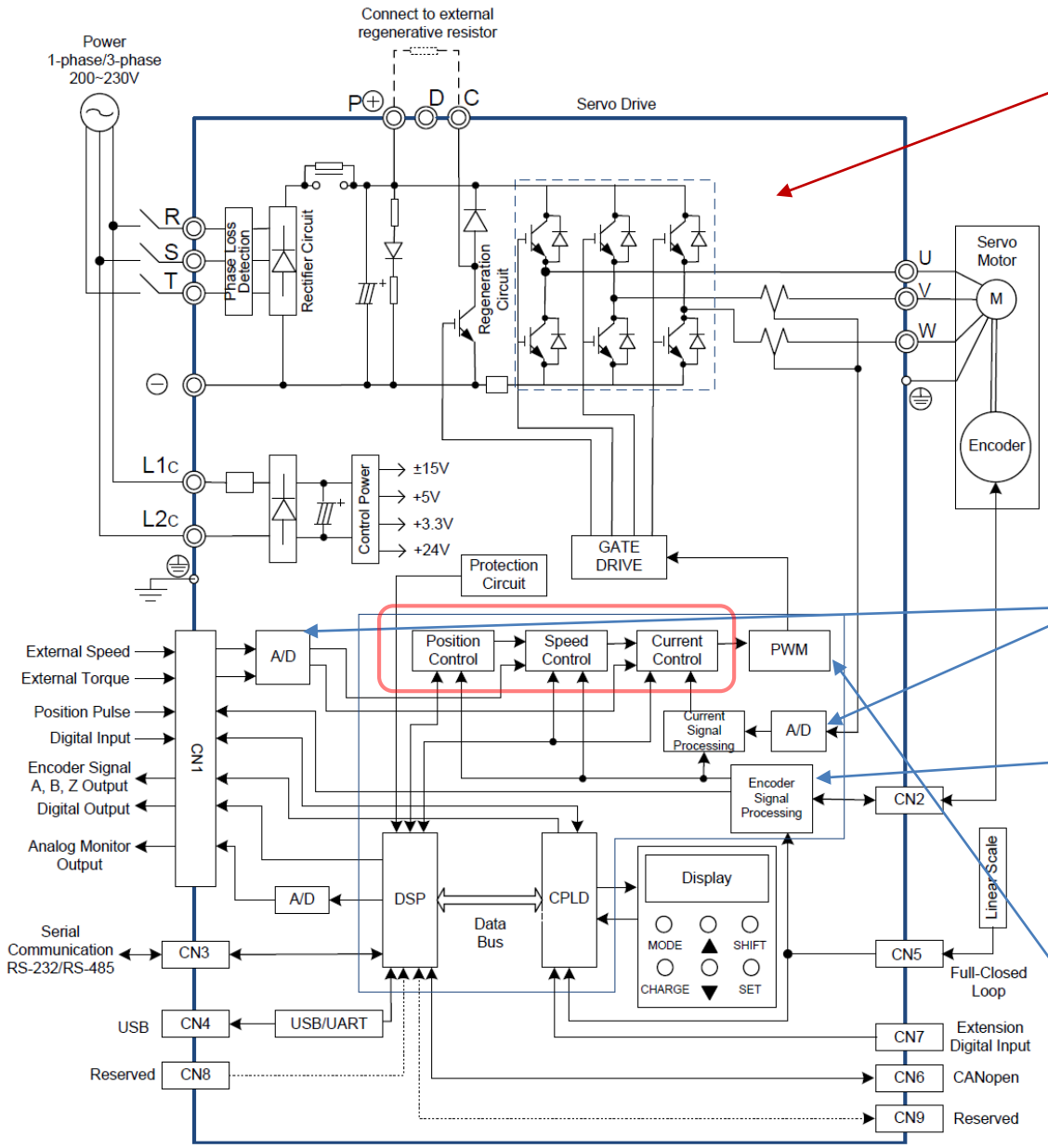
Consumo: Modos de bajo consumo (en aplicaciones portátiles, registradores etc)

Condiciones ambientales: Rango de temperatura, inmunidad a ruido etc.

Más allá de los aspectos técnicos está la factibilidad y la economía. Evaluar disponibilidad (presente/futura) de los componentes, de **herramientas de desarrollo**, y si el **encapsulado** permite prototipar.

Los fabricantes de semiconductores ofrecen familias de µC a partir de un núcleo (CPU), con más pines, periféricos, RAM o ROM, de forma que si un µC nos queda pequeño en alguno de estos recursos, se puede pasar a uno similar con dicho recurso ampliado.

Ejemplo: Servocontrolador industrial



La etapa de potencia está constituida por un rectificador de alta tensión (de 220V monofásicos o trifásicos) y un puente trifásico, de transistores MOSFET ó IGBT.

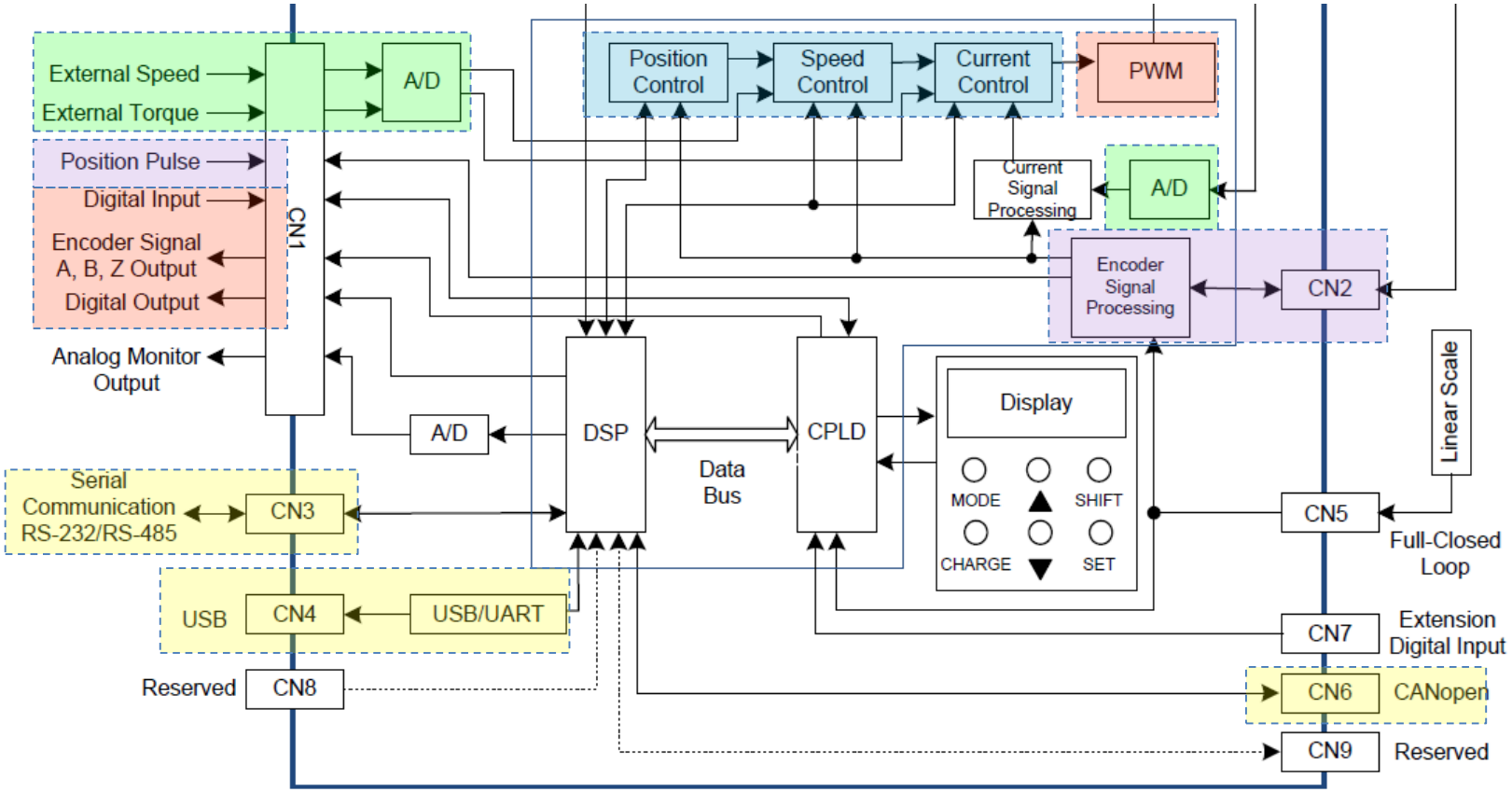
El hardware de procesamiento incluye núcleos DSP (digital signal processing), CPLD (lógica programable) y módulos de E/S para medición, generación de señales y comunicaciones:

Las **entradas analógicas** del µC miden corriente (para determinar torque), voltaje (para detectar fuerza contra-electromotriz o *Back EMF*) y consignas de Velocidad y/o Torque dadas en forma de tensión.

Las entradas de **QEI**, **Timers** conectados en modo contador o medición precisa de tiempos, y **entradas de captura** para sensores Hall permiten determinar directa o indirectamente posición y velocidad del eje, para realizar su control

Las salidas del µC, directas o moduladas (**PWM**) conmutan, a través de "*gate drivers*", los transistores que comandan el motor.

Ejemplo: Servocontrolador industrial (2)



Entradas A/D para medir corriente (para determinar torque), voltaje (para detectar fuerza contraelectromotriz) y consignas de Velocidad y/o Torque dadas como tensión.

Salidas **PWM** (modulación de ancho de pulso) o de propósito general para excitar puente trifásico.
Líneas **GPIO** (E/S de propósito general).

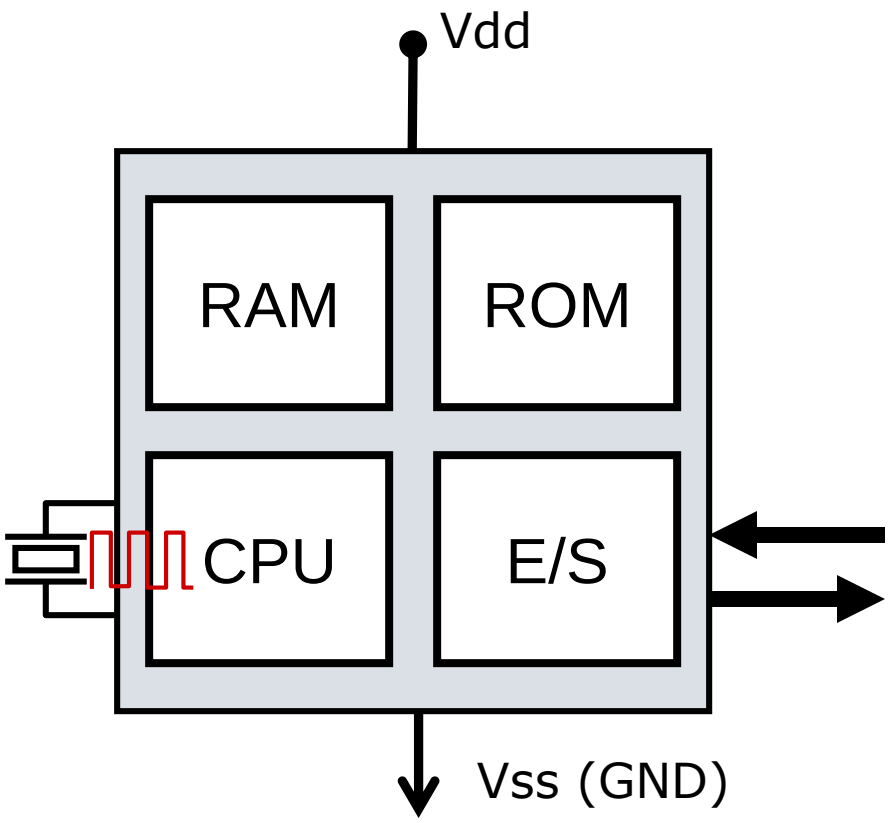
Entradas de **QEI** (interfaz de encoder en cuadratura)
Medición precisa de tiempo para calcular velocidad.

Interfaces de comunicación **interna** (entre subsistemas como memorias, sensores, conversores D/A etc).
Interfaces de comunicación **externa** (protocolos estándar)

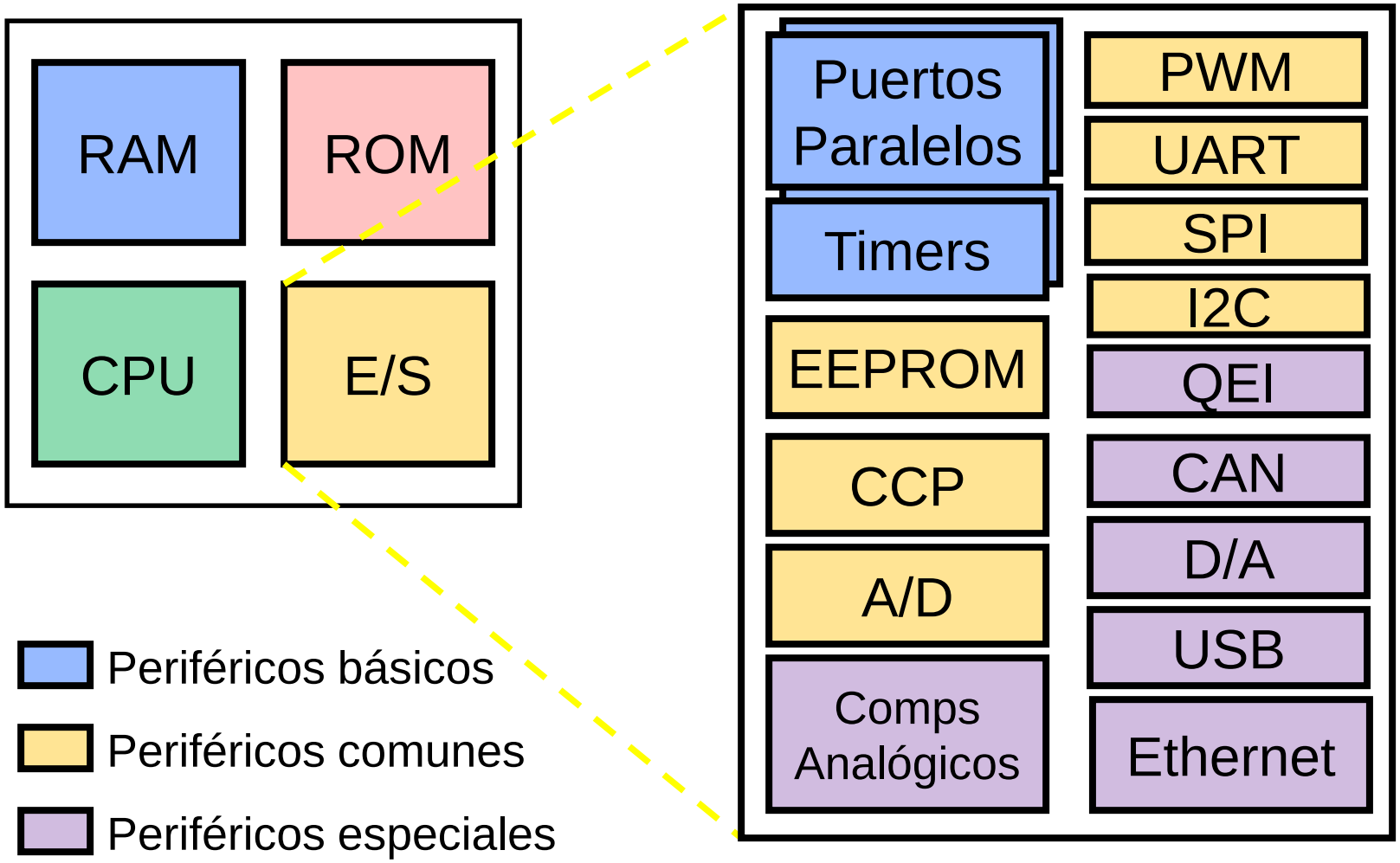
Ejecución de lazos de control en paralelo con temporización precisa (comportamiento **determinístico**, posible **RTOS**)

Capacidad de procesamiento según complejidad de cómputo de los algoritmos

Microcontrolador (µC)



Bloques de un microcontrolador (2)



Hojas de datos de microcontrolador

Resumen de características: Arquitectura, velocidad, memorias, periféricos, consumo etc.
Encapsulados, pines: resumen de funciones de cada pin.
Estructura interna: Arquitectura
Organización de la memoria: De datos y programa. Mapa de registros especiales
Detalle de puertas de E/S: Lógica asociada a los pines de E/S en sus distintas funciones.
Configuraciones de oscilador y modos de trabajo.
Subsistemas periféricos: Timers, conversores A/D, UART, PWM etc. Registros asociados.
Set de instrucciones: Resumen de operaciones ASM, ciclos, bits de SR afectados.
Especificaciones eléctricas, de grabado etc.

Variantes en microcontroladores

Modos de oscilación

- Fuentes para obtener el reloj de sistema
- Relación entre Frecuencia de Oscilador y Frecuencia de Ciclo de Instrucción

Modos de RESET

- Power-On. Brown-Out. Watchdog

Modos de consumo.

Modos de Interrupción

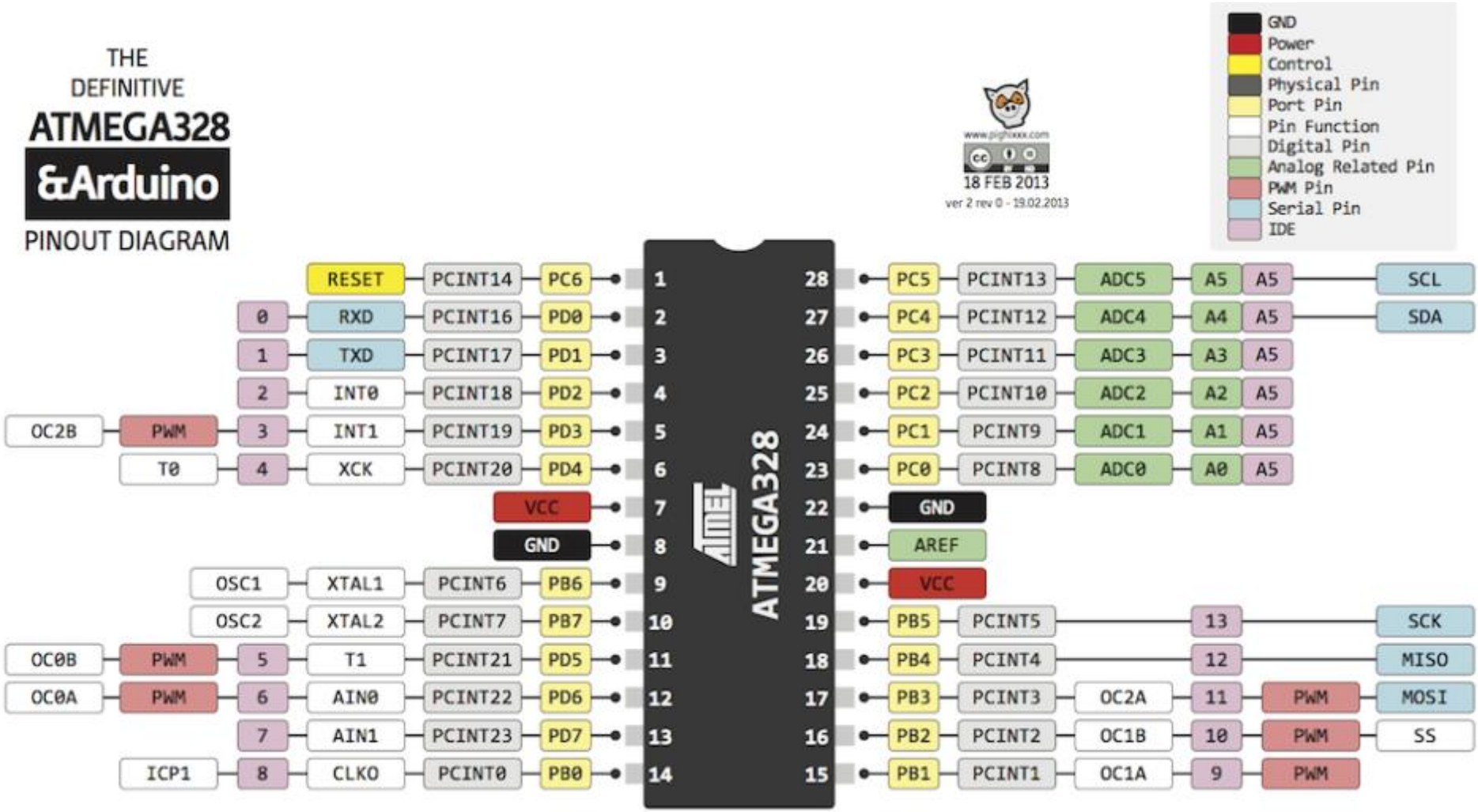
- Vector/es de interrupción
- Interrupciones anidadas/concurrentes
- Interrupciones no enmascarables

Organización de Memoria

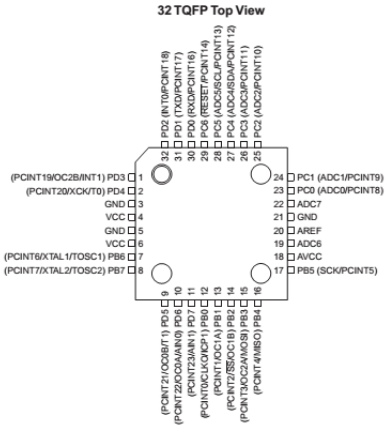
- Harvard-Paginado/segmentada (Ej. Intel 8051, Microchip PIC16/PIC18)
- Harvard-Direccionamiento lineal (Ej. Micros AVR de Atmel, micros de 16/32/64 bits en general)
- Von Neumann. (Ej MSP430 de Texas , 68HCxx de Motorola etc)
- Pila dedicada (PICs), pila en RAM (demás)

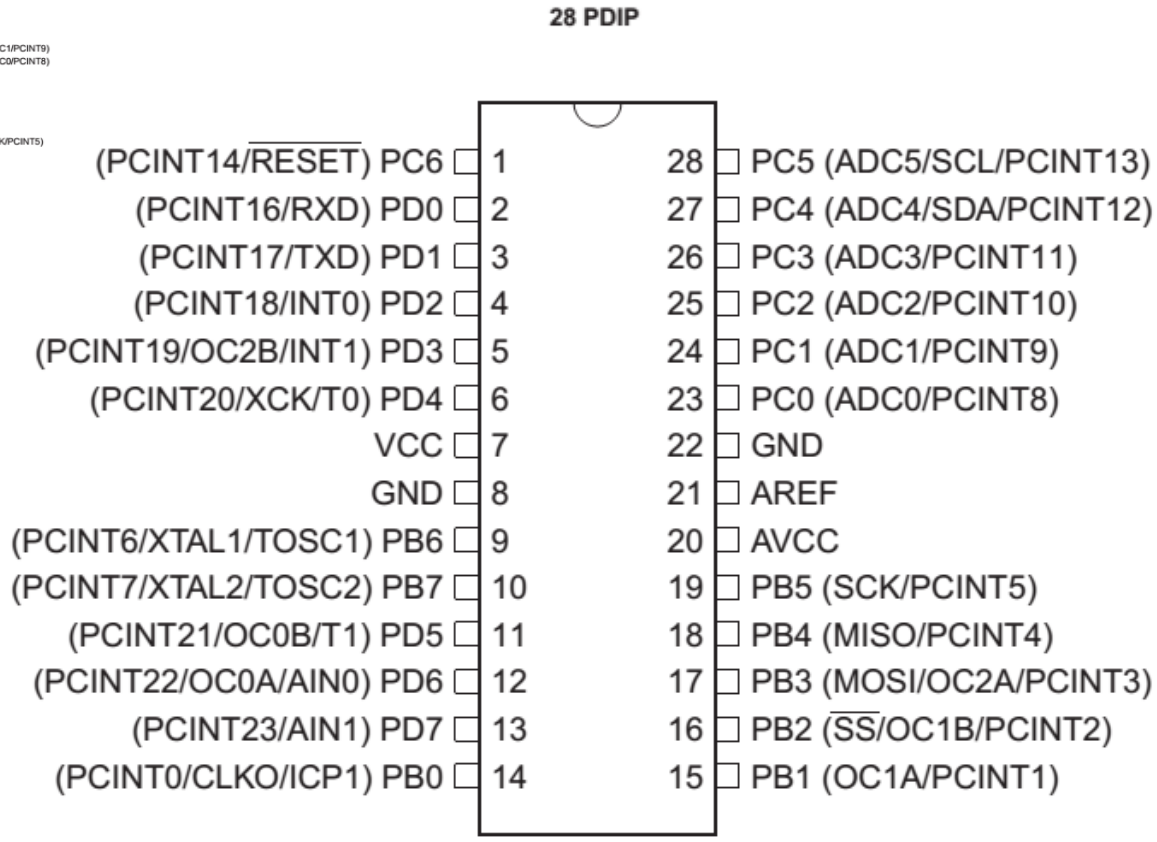
Set de instrucciones: Reducido, Extendido. Ortogonal/no ortogonal.

ATmega328 en Arduino

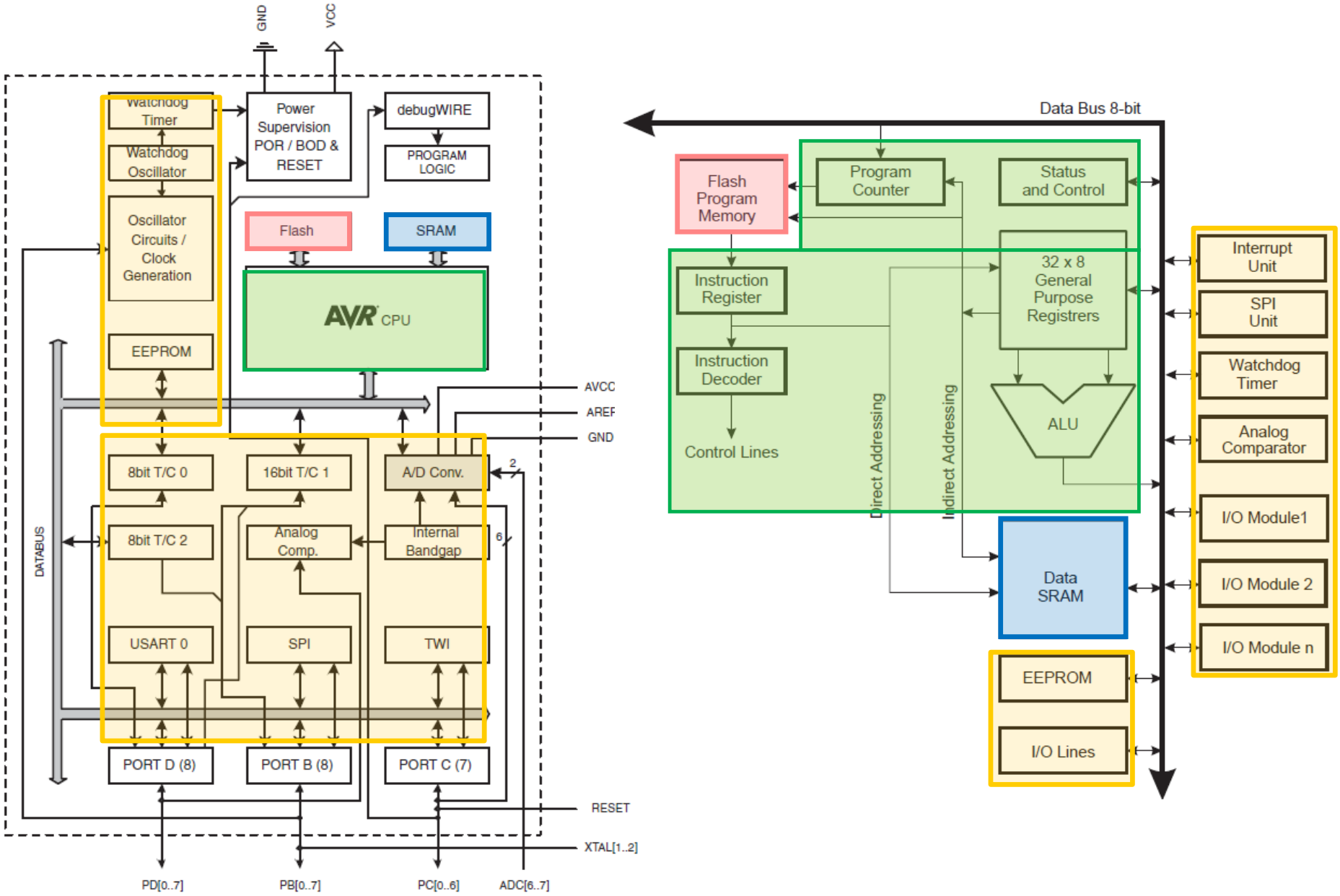


ATmega328 (hoja de datos)

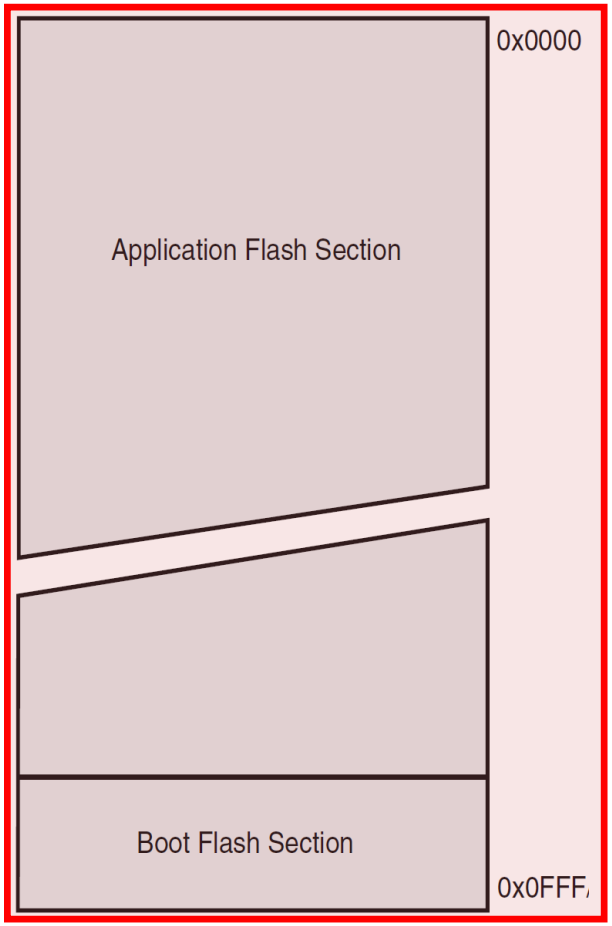




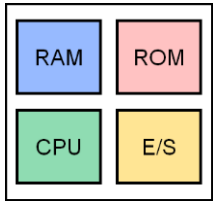
Arquitectura de un AVR (ATmega48 a ATmega328)



Organización de las memorias (Ej. ATmega328)

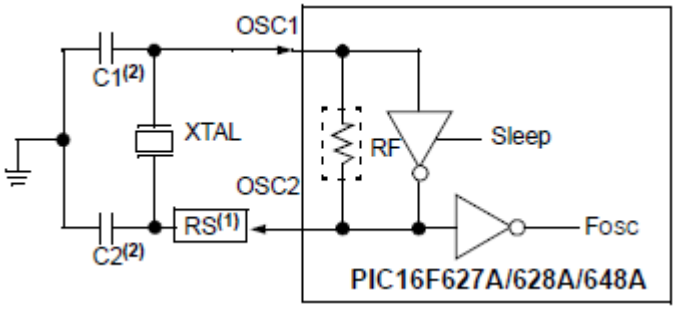


32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Reg.	0x0060 - 0x00FF
Internal SRAM (512/1024/1024/2048 x 8)	0x0100 … 0x02FF/0x04FF

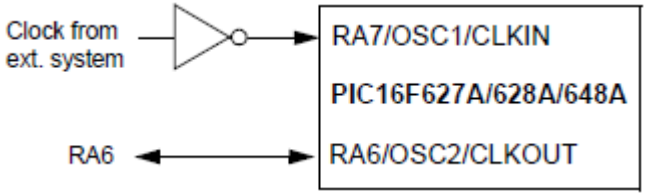


Modos de oscilación básicos

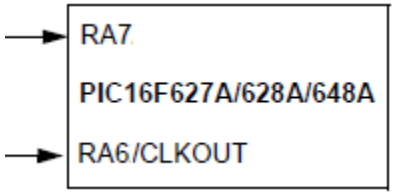
XT (cristal)
LP (*low power crystal*)
HS (*high speed crystal*)



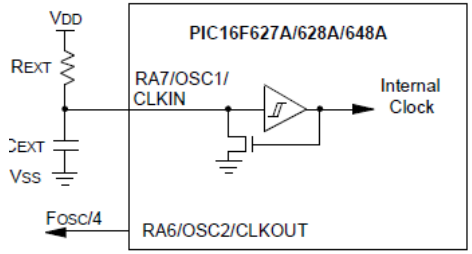
EC *External clock in*

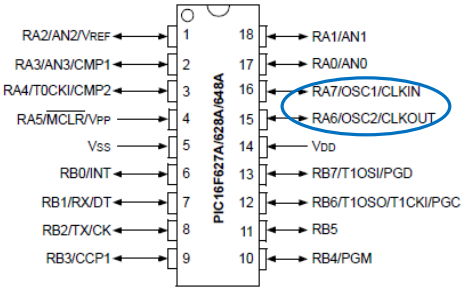


INTOSC

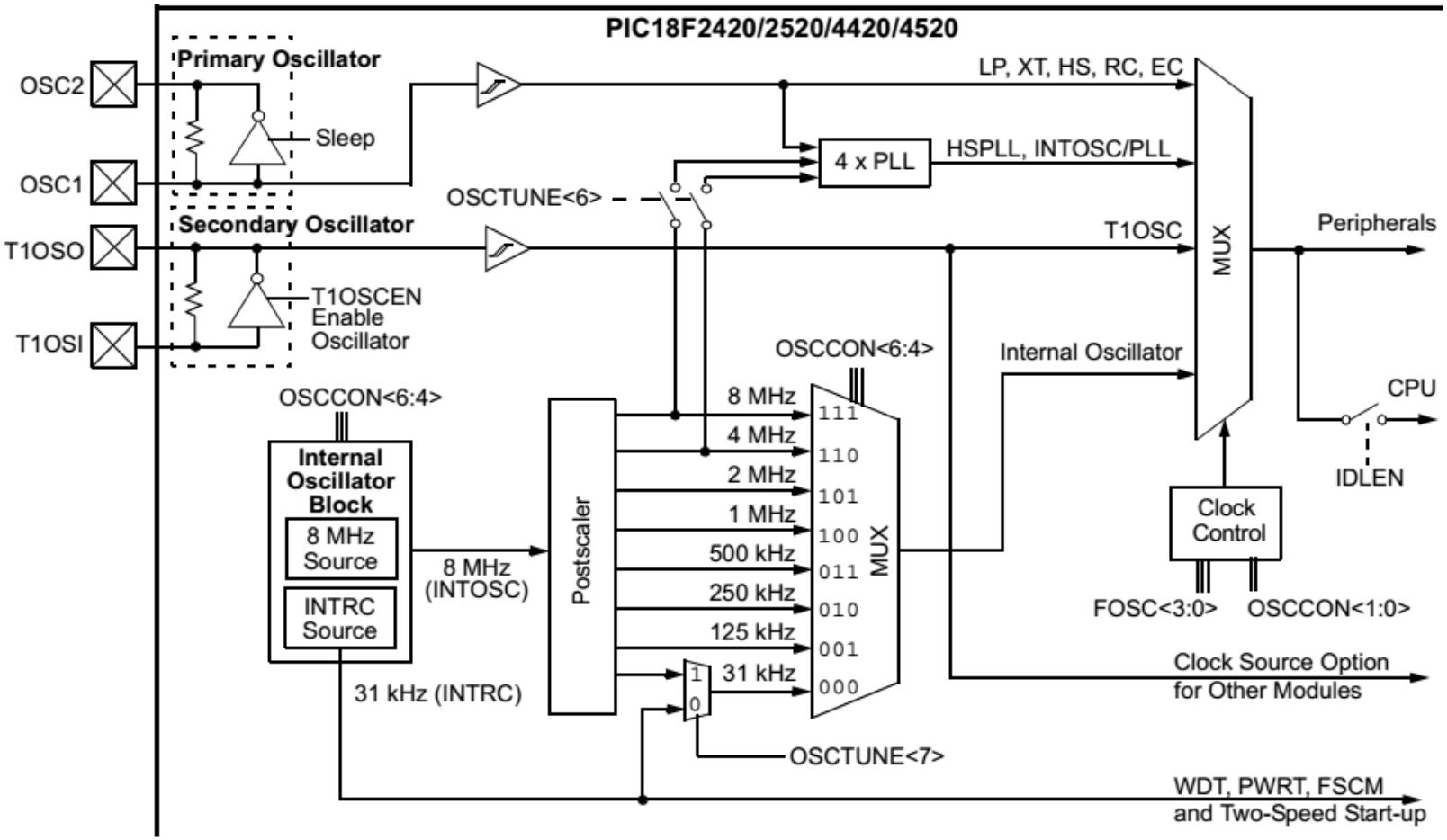


RC



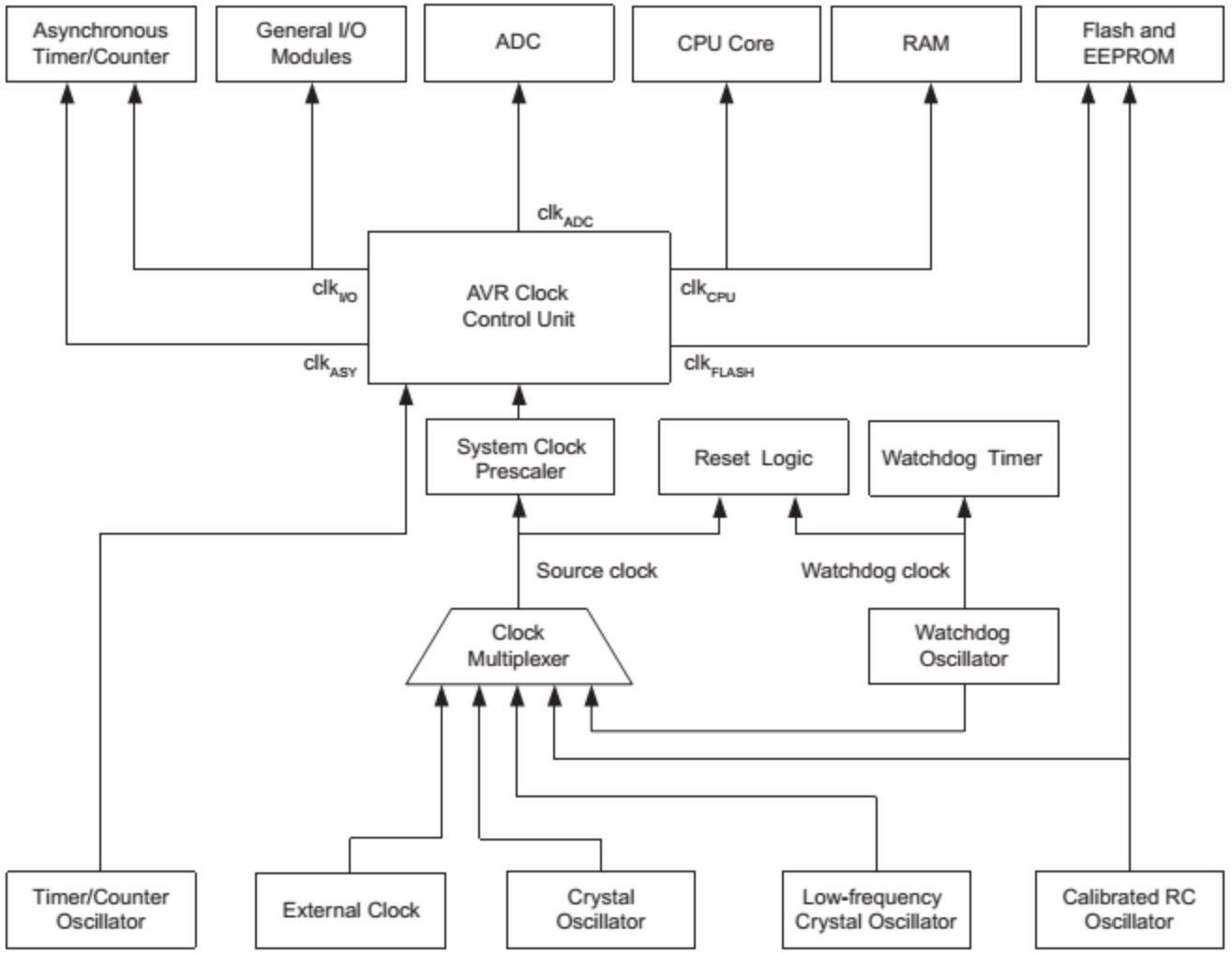


Fuentes de reloj (PIC18F, otros)

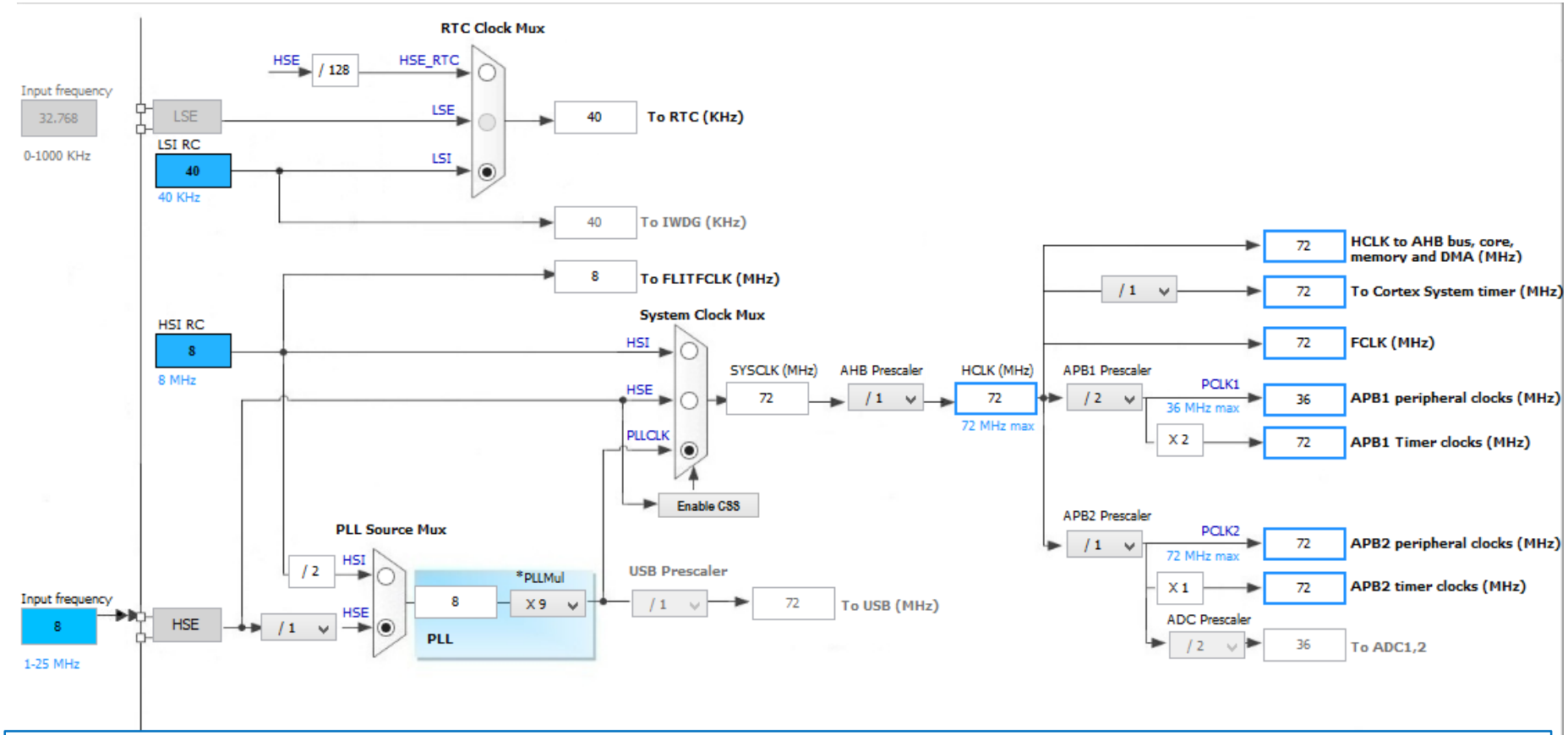


En microcontroladores de gama media/alta se dispone de circuitos PLL (*Phase Locked Loop*) para multiplicar el reloj externo por un factor entero o fraccional.
También se dispone de oscilador secundario (normalmente para baja frecuencia, decenas de kHz) para modos de muy bajo consumo, y oscilador independiente de baja frecuencia para el ***watchdog***

Fuentes de reloj (AVR)



Fuentes de reloj (ARM Cortex M3)



Clock tree del STM32F103C8T6. Dispone de PLL y *prescalers* para obtener las frecuencias adecuadas para cada subsistema. La fuente principal puede ser externa (HSE: clock externo, cristal, resonador cerámico) o interna (HSI: oscilador RC de precisión típica 1%).
Para el watchdog la fuente es un oscilador RC interno independiente de 40 kHz.
La configuración debe procurar no exceder las frecuencias máximas admisibles en distintas etapas de la cadena de clock. Existen herramientas como el STMCubeMX (para micros de ST), que facilitan esta configuración.

Juego de instrucciones – AVR (parcial)

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z, C, N, V, H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z, C, N, V, H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z, C, N, V, S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z, C, N, V, H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z, C, N, V, H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z, C, N, V, H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z, C, N, V, H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z, C, N, V, S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z, N, V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z, N, V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z, N, V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z, N, V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z, N, V	1
COM	Rd	One's Complement	Rd ← $FF − Rd	Z, C, N, V	1
NEG	Rd	Two's Complement	Rd ← $00 − Rd	Z, C, N, V, H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z, N, V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • ($FF − K)	Z, N, V	1
INC	Rd	Increment	Rd ← Rd + 1	Z, N, V	1
DEC	Rd	Decrement	Rd ← Rd − 1	Z, N, V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z, N, V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z, N, V	1
SER	Rd	Set Register	Rd ← $FF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd × Rr	Z, C	2
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd × Rr	Z, C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd × Rr	Z, C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← Rd × Rr << 1	Z, C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← Rd × Rr << 1	Z, C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← Rd × Rr << 1	Z, C	2
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2

BIT AND BIT-TEST INSTRUCTIONS					
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z, C, N, V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z, C, N, V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z, C, N, V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z, C, N, V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z, C, N, V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow.	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1

Ver sección 34 en el manual de micros AVR

3.B
Programación

Estructura básica de un programa de µC

CONFIGURACIÓN

Elegir los pines E/S de acuerdo a un esquema y los periféricos a utilizar en la aplicación (timers, puerto serie etc).

Escribir los registros de activación y configuración de E/S y periféricos

INICIALIZACIÓN

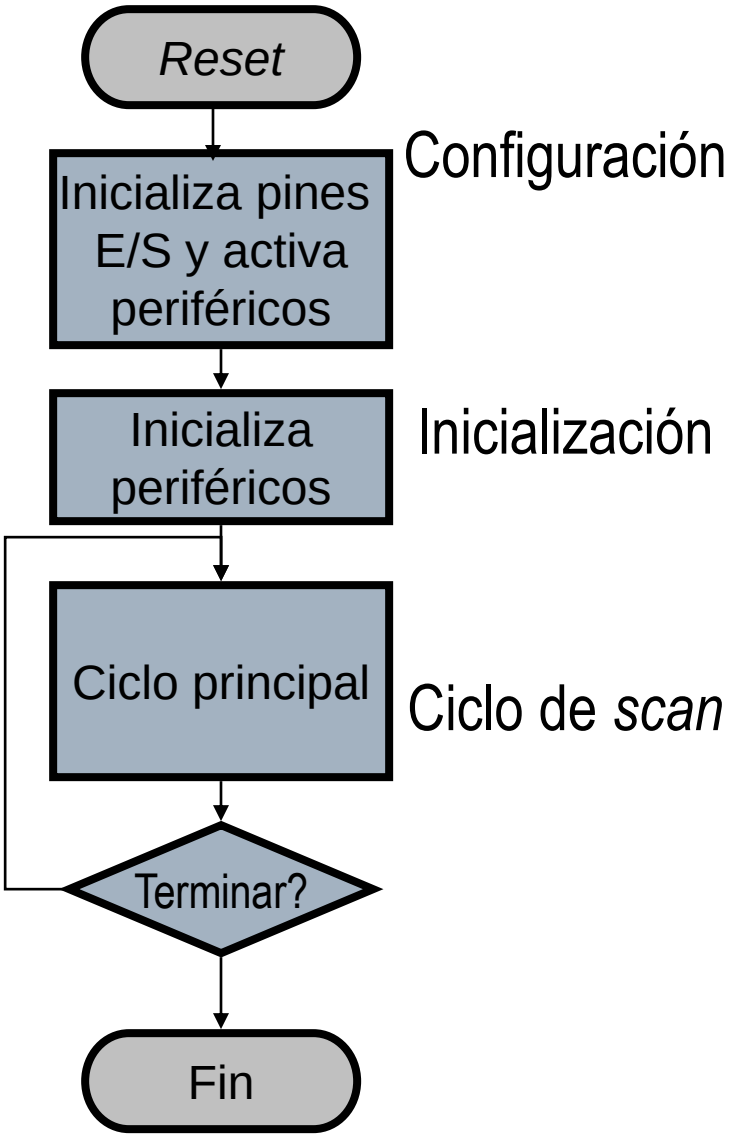
Poner los periféricos y las E/S en un estado inicial adecuado (ej. las salidas en '0')

CICLO DE SCAN

Es el programa principal, que por lo general se ejecuta cíclicamente.

Puede estar complementado con rutinas de servicio de interrupción (ISRs). Esta arquitectura se denomina *foreground/background*, donde el ciclo principal es el *background* y la ISRs pasan al *foreground*. cuando son llamadas.

En las arquitecturas basadas en tareas (ej RTOS) el "**ciclo de scan**" puede estar vacío y todas las funciones se realizan en tareas (tasks) fuera del main.



Lenguajes de Programación de µC

ALTO NIVEL (C, C++ etc)	Ensamblador (*assembler*)
• Requiere compilador	• Traducción directa a lenguaje de máquina.
• Código portable	• Específico del Hardware
• Comprensible. Facilidad para trabajar programas grandes, operaciones aritméticas y de formateo de datos.	• Difícil seguimiento y concepción de programas grandes
• Los IDEs incluyen bibliotecas de funciones para manejo de periféricos.	• Debe manipularse los registros que controlan cada periférico.
• Menos eficiente en velocidad y en tamaño del código (aunque los compiladores ofrecen niveles de optimización).	• Puede optimizarse en velocidad y tamaño

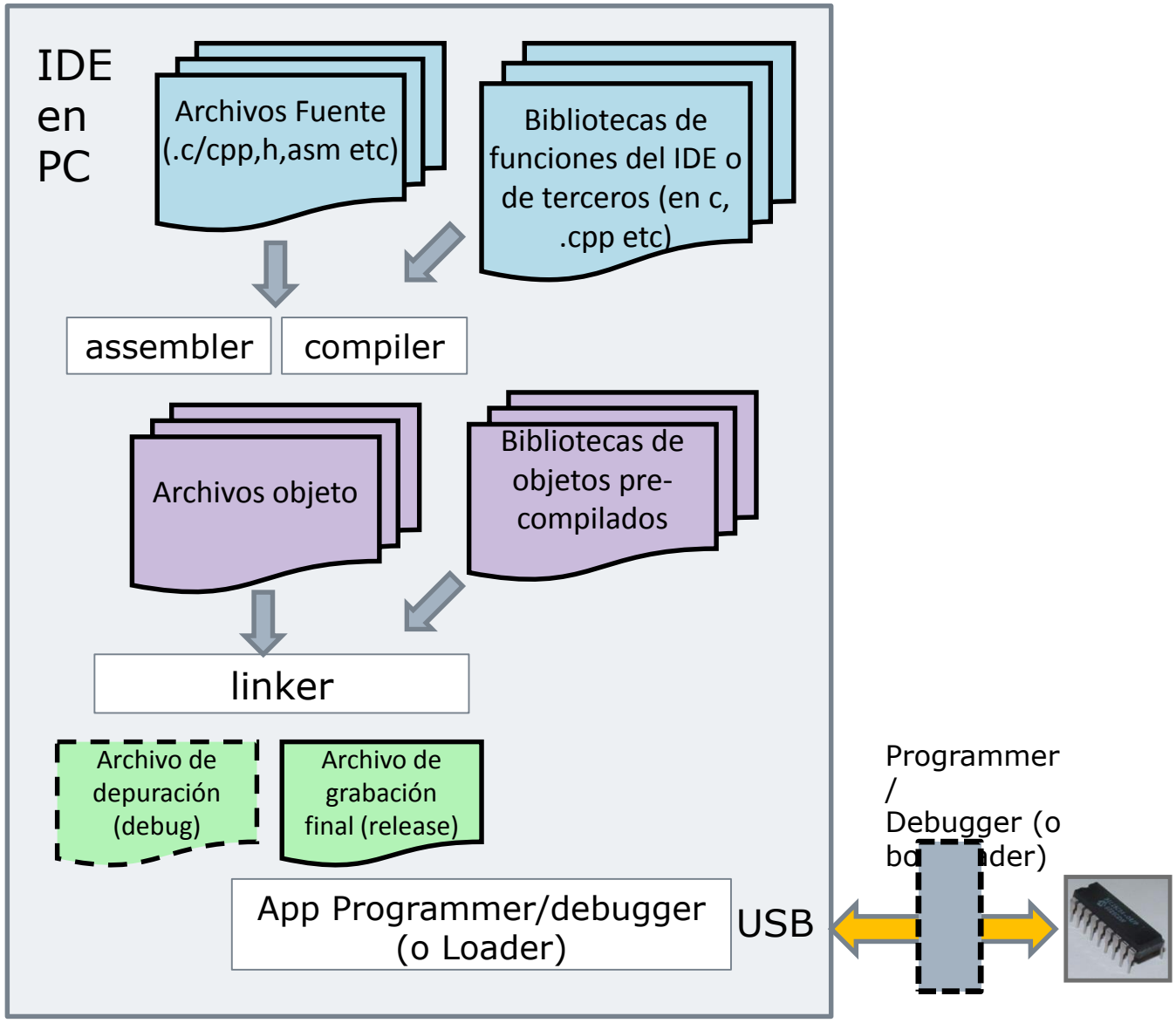
Se suele utilizar lenguaje de alto nivel, con rutinas críticas en assembler

Compilación-grabación (esquema general)

```
do
   {               // llave abr
   A=atoi(getc()); //recibe dat
   B=atoi(getc()); //recibe dat
   C=A+B;          //suma A y B
   printf("La suma es: %d",C);
   }while(C!=0);   // el ciclo
```

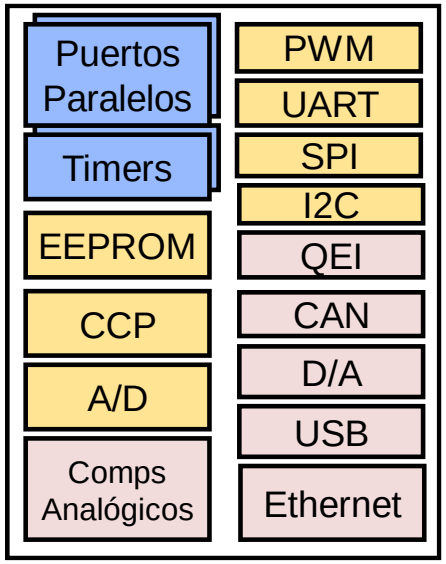
```
01DD: MOVWF   0x2A
01DE: CALL    0068
01DF: MOVF    0x78,W
01E0: MOVWF   0x27
01E1: MOVF    0x27,W
01E2: ADDWF   0x26,W
01E3: MOVWF   0x28
01E4: CLRF    0x29
01E5: MOVF    0x29,W
```

```
0A1010130A200A301A401
0C039083881008312900
0003083169200073083129
01030F700F70BCA2900008
083169F01831217207808
07808A60017207808A900
0A70027082607A800A901
0F700AC0027210C302902
0AA001830AB006229A808
```



Un buen resumen del proceso y tipos de archivo en el siguiente documento

3.C
Subsistemas de un µC



Conceptos preliminares

Todo subsistema o **periférico** de un µC (TIMER, UART, A/D, GPIO etc) se puede activar/desactivar, configurar para que funcione de cierto modo (velocidad, resolución, período etc) e intercambiar datos. También se puede verificar su estado (dato listo, vacío, lleno etc).

El manejo del subsistema 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. Hay registros que tienen bits de Estado y de Control, especialmente en micros en los que 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. Un ejemplo de conjunto de bits: los 4 bits de selección de canal de un multiplexor 16:1 conectado a un módulo A/D.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

ADCON0: A/D CONTROL REGISTER 0 en PIC16F884

Manipulación de bits (*)

1) Operadores de desplazamiento: **>>** y **<<**

b = a << 2; ***//*** **desplazamiento a izquierda**

Ejemplo: si a y b son enteros de 8 bits

Si a = 10101101

resulta b = 10110100

Esta operación en un entero sin signo equivale a multiplicar por 2^2

b = a >> 3; ***//*** **desplazamiento a derecha**

Si a = 10101101

resulta b = 00010101

Esta operación en un entero sin signo equivale a dividir por 2^3

2) Operador inversor: **~** (virgulilla o tilde)

b = ~a; ***//*** **inversor de bits**

Ejemplo: si a y b son enteros de 8 bits

Si a = 10101101

resulta b = 01010010

Nota: No confundir ~ con el NOT booleano (!)

(*)Ver en página de cátedra Operaciones bit a bit (bitwise)

Manipulación de bits (2)

3) Operadores **|** **&** y **^** **(***bitwise* **and, or, xor)**

c = a | b; *//* realiza la OR bit a bit entre a y b

Ejemplo: si a y b son enteros de 8 bits

```
Si  a = 10101101
    b = 00000110
resulta
    c = 10101111
```

c = a & b; *//* realiza la AND bit a bit entre a y b

Ejemplo: si a y b son enteros de 8 bits

```
Si  a = 10101101
    b = 11110111
resulta
    c = 10100101
```

c = a ^ b; *//* realiza la XOR bit a bit entre a y b

```
Si  a = 10101101
    b = 00000110
resulta
    c = 10101011
```

Manipulación de bits: Ejercicios

Exprese el valor de A en binario:

```
A = (1<<4);
A = (1<<0)|(1<<5);
A = ~(1<<3);
A = ~((1<<0)|(1<<5));
```

¿Qué le sucede a la variable A?

```
A |=(1<<4); //equivale a A = A|(1<<4);
A &=(1<<4); //equivale a A = A & (1<<4);
A &=~(1<<4);//equivale a A = A & ~(1<<4);
A ^=(1<<4); //equivale a A = A^(1<<4);

#define PULS 3
A |=(1<<PULS) ;
```

(¿Cuánto vale A en este caso?)

Manipular los bits solicitados (sin alterar el resto de los bits)

- Poner en ‘1’ el bit 3 de un variable PuertoB
- Poner en ‘0’ el bit 4 de una variable PuertoC
- Invertir el bit 2 de una variable PuertoA
- Verificar con un “if” si el bit 5 de una variable PuertoB es ‘1’ o ‘0’
- Poner en ‘1’ los bits 2, 4 y 5 de una variable PuertoC
- Colocar en “011” los bits consecutivos 6,5,4 de una variable PuertoA

Entradas/Salidas de propósito general (GPIO)

Los pines de un µC permiten la interconexión de sus sistemas internos con dispositivos exteriores. Estos pines están agrupados en puertos, comúnmente de 8 bits (en otros pueden ser 16 o 32), y se accede a ellos mediante registros mapeados en memoria.
La función **básica** o de **propósito general** consiste en leer los valores lógicos de estos pines, en forma individual o agrupada, o en escribirles valores lógicos también de forma individual o agrupada.
Normalmente los pines se configuran al inicio del programa como entrada o salida escribiendo en bits de dirección asociados.
En los **microcontroladores PIC**, los pines individuales de un puerto PORTx son configurados como Entrada o Salida escribiendo respectivamente **'1' ó '0'** en los bits correspondientes del registro TRISx. Por ejemplo,

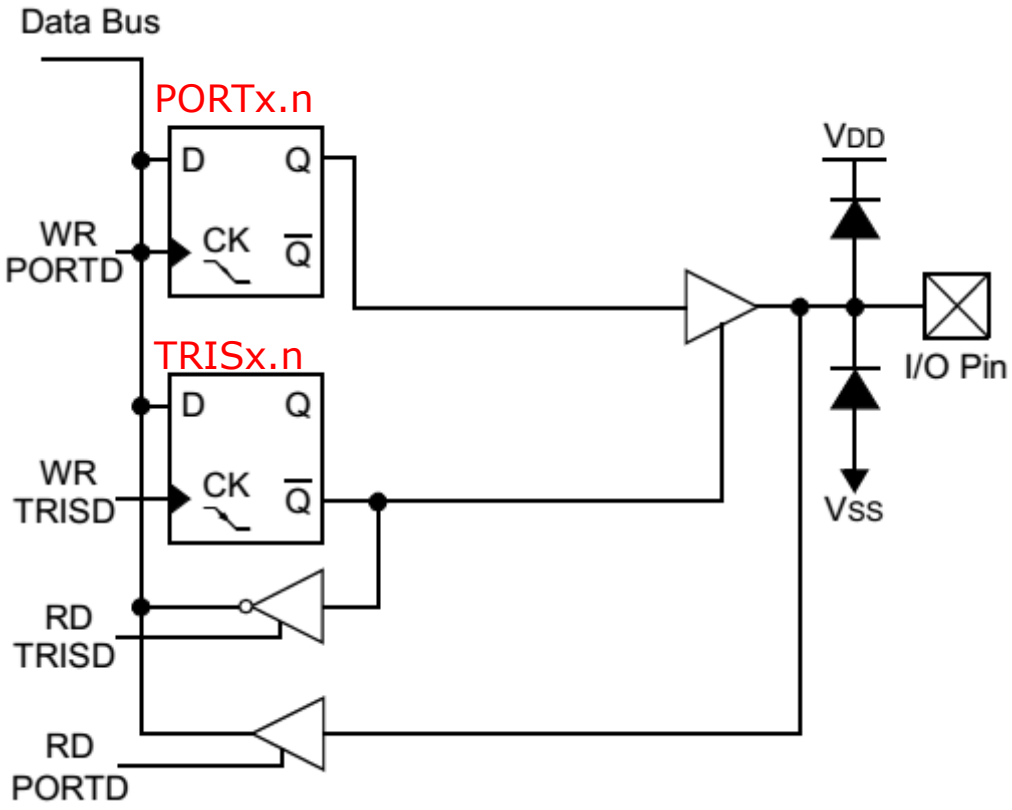
Si se quiere	PORTB	b7	b6	b5	b4	b3	b2	b1	b0
		S	E	E	E	S	S	E	S	(S: salida E: entrada)
Debe ser	TRISB	b7	b6	b5	b4	b3	b2	b1	b0
		0	**1**	**1**	**1**	**0**	**0**	**1**	**0**

En los **microcontroladores ATmega**, los pines individuales de un puerto PORTx son configurados como Entrada o Salida escribiendo respectivamente **'0' ó '1'** en los bits correspondientes del registro DDRx (*Data Direction Register* X). Por ejemplo

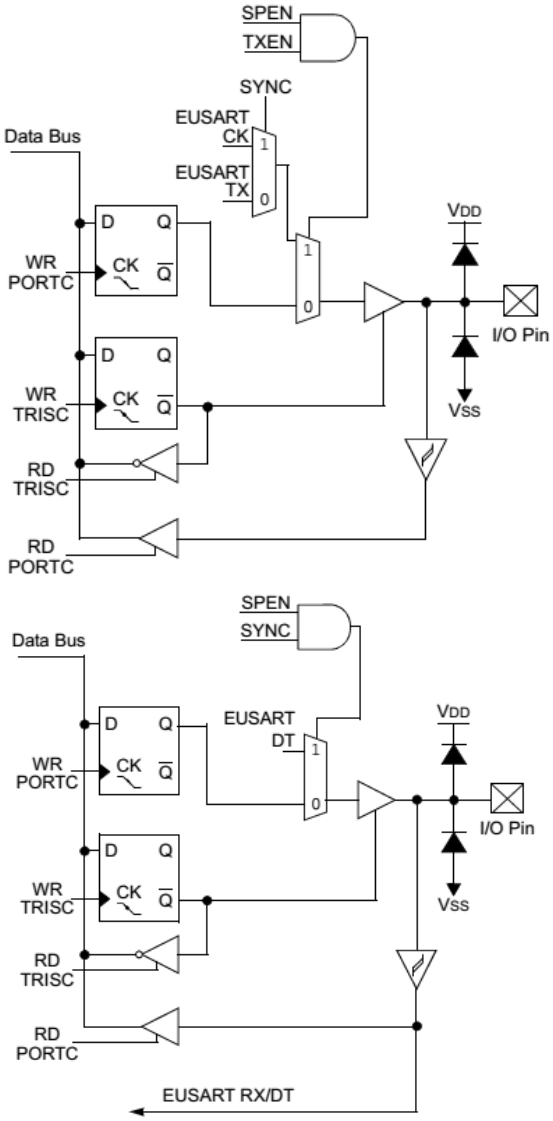
Si se quiere	PORTB	b7	b6	b5	b4	b3	b2	b1	b0
		S	E	E	E	S	S	E	S	(S: salida E: entrada)
Debe ser	DDRB	b7	b6	b5	b4	b3	b2	b1	b0
		1	**0**	**0**	**0**	**1**	**1**	**0**	**1**

Circuito asociado a un pin de E/S (GPIO) en PIC

E/S típica (16F)

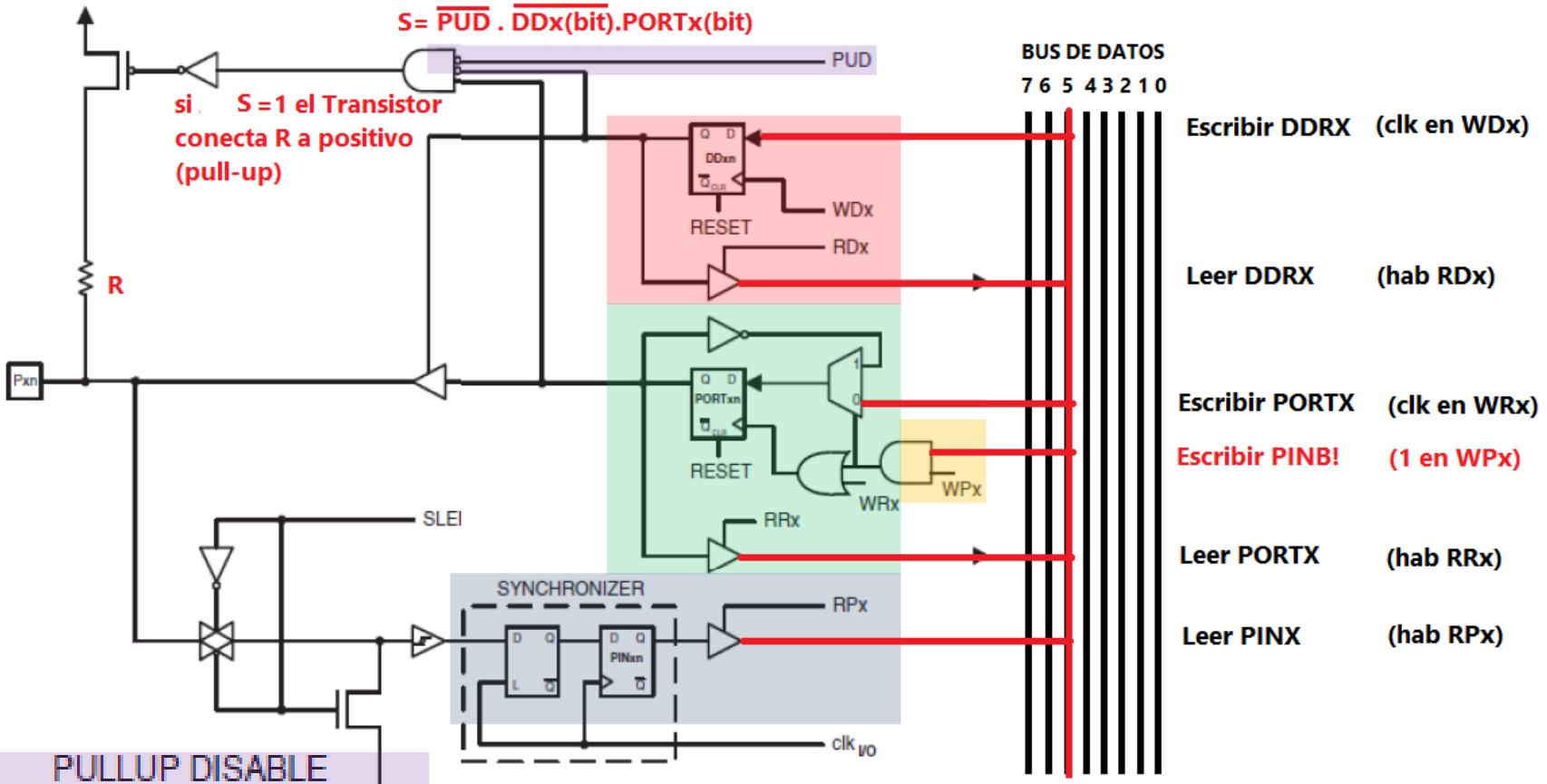


Pin compartido con otras funciones



El valor de PORTx.n actúa sobre el pin si TRISx.n es ‘0’.

Circuito asociado a un cada pin de E/S (GPIO) en AVR



PUD:	PULLUP DISABLE
SLEEP:	SLEEP CONTROL
$clk_{I/O}$:	I/O CLOCK
WDx:	WRITE DDRx
RDx:	READ DDRx
WRx:	WRITE PORTx
RRx:	READ PORTx REGISTER
RPx:	READ PORTx PIN
WPx:	WRITE PINx REGISTER

Registros para GPIO en AVR

En los microcontroladores **ATmega**, los pines están agrupados en puertos de 8 bits. Cada puerto "x" tiene 3 registros asociados:

DDRx (Data Direction Register) para definir el sentido de circulación de los datos en cada pin (es decir si va a ser Entrada o Salida). Ejemplo: DDRA, DDRB etc

PORTx : Lo que se escribe en este registro se presenta en los pines de salida. Ej. PORTA, PORTB etc. *¡Cuidado! si un pin ha sido definido como entrada y se escribe un '1' en el bit correspondiente del registro PORTx, se le habilita la resistencia de pull-up, de unos 100kohms. Es decir se produce un '1' débil en el pin aunque esté configurado como entrada. (Para evitarlo debe escribirse un '1' en el bit PU del registro MCUCR)*
PINx: Leyendo este registro se leen los pines de entrada.

Ejemplo:

```
DDRB=(1<<3)|(1<<5);    // Hace PB3 y PB5 salidas, los demás entradas
PORTB =0b11111111;     // Hace PB3=1, PB5=1 (las entradas quedan con pull-up!)

#define ACT (PINB&(1<<3))  // ACT valdrá 0 sin PINB3=0 y distinto de 0 si PINB3=1

if (ACT)…  // puede servir para realizar una acción según ACT sea verdadero (!=0) o falso (==0)
```

Actividad

- Configurar entorno de desarrollo: Atmel Studio o WinAVR
- Conectar placa Arduino e instalar driver de comunicación USB (FT232 o CH340)
- Verificar puerto USB. Cambiar en argumentos de avrdude.
- Probar ejemplos.
- Realizar ejercicios propuestos.

GPIO: Ejercicios

Ejercicio 1:
Realizar un programa en ATmega328P con salida en PB5. La salida PB5 debe producir una señal cuadrada de 2 Hz permanente.

```
#define F_CPU 16000000
#include <avr/io.h>
#include <util/delay.h>
```

Ejercicio 2:
Realizar un programa en ATmega328P con dos **salidas** en **PB5 y PB2**, **entrada** en **PB3**. La salida PB5 debe producir una señal cuadrada de 2 Hz permanente, y la salida en PB2 debe invertirse cada vez que se pulse PB3.

Ejercicio 3:
Realizar un programa en ATmega328P con **salida** en **PB5**, y dos **entradas** en **PB3 y PB4**. El autómata tendrá dos estados, Activado y Desactivado. Inicialmente estará Activado, y en este estado deberá producir una señal de 1Hz, Duty Cycle 70% en PB5. Al pulsar PB4 debe pasar a modo Desactivado, y en tal estado poner la salida en 0. Al pulsar PB3 debe volver al estado Activado y generar la señal de 1Hz.

Interrupciones

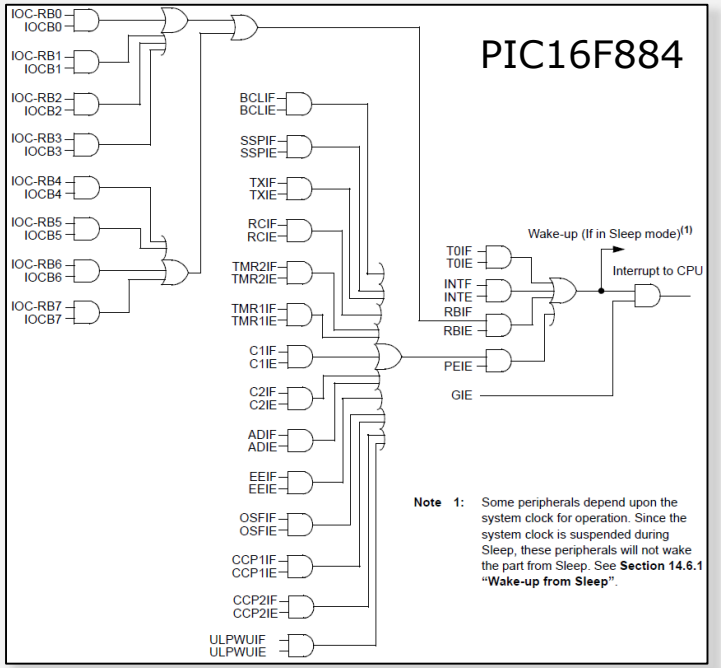
Las interrupciones en los microcontroladores permiten atender de manera inmediata a los distintos periféricos que pudieran requerirlo (puerto de comunicaciones, *timers*, puerto paralelo, conversor A/D etc).

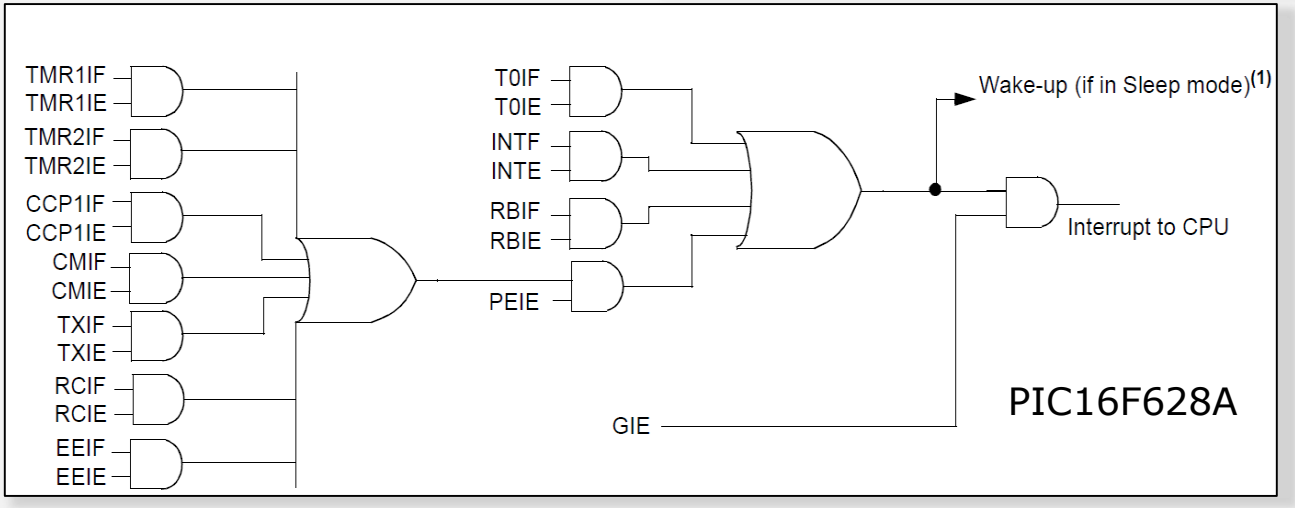
Las interrupciones de periféricos pueden ser **habilitadas** o **enmascaradas** por programa, y son **disparadas** por los eventos que pudiera generar cada periférico. Por ejemplo, el fin de una conversión A/D, la recepción de un byte por un puerto serie, un flanco de subida (o bajada) en un pin específico, el cambio de un bit en un puerto, el desborde de un contador interno (*timer*).
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 un µC 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 entonces, para el periférico que lo necesite, una **rutina de servicio**, que consiste en instrucciones para procesar ese evento (leer el dato convertido o recibido, atender el sistema externo que produjo el flanco en INT0 etc).
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.
Lo deseable es que el programa se mantenga en sus tareas habituales y salte a la rutina de servicio del periférico cuando este lo demande.
El software puede selectivamente habilitar una o más de estas fuentes de interrupciones.
Algunas interrupciones especiales son **no enmascarables**.

(*) La denominación de los flags es propia de cada familia de microcontrolador

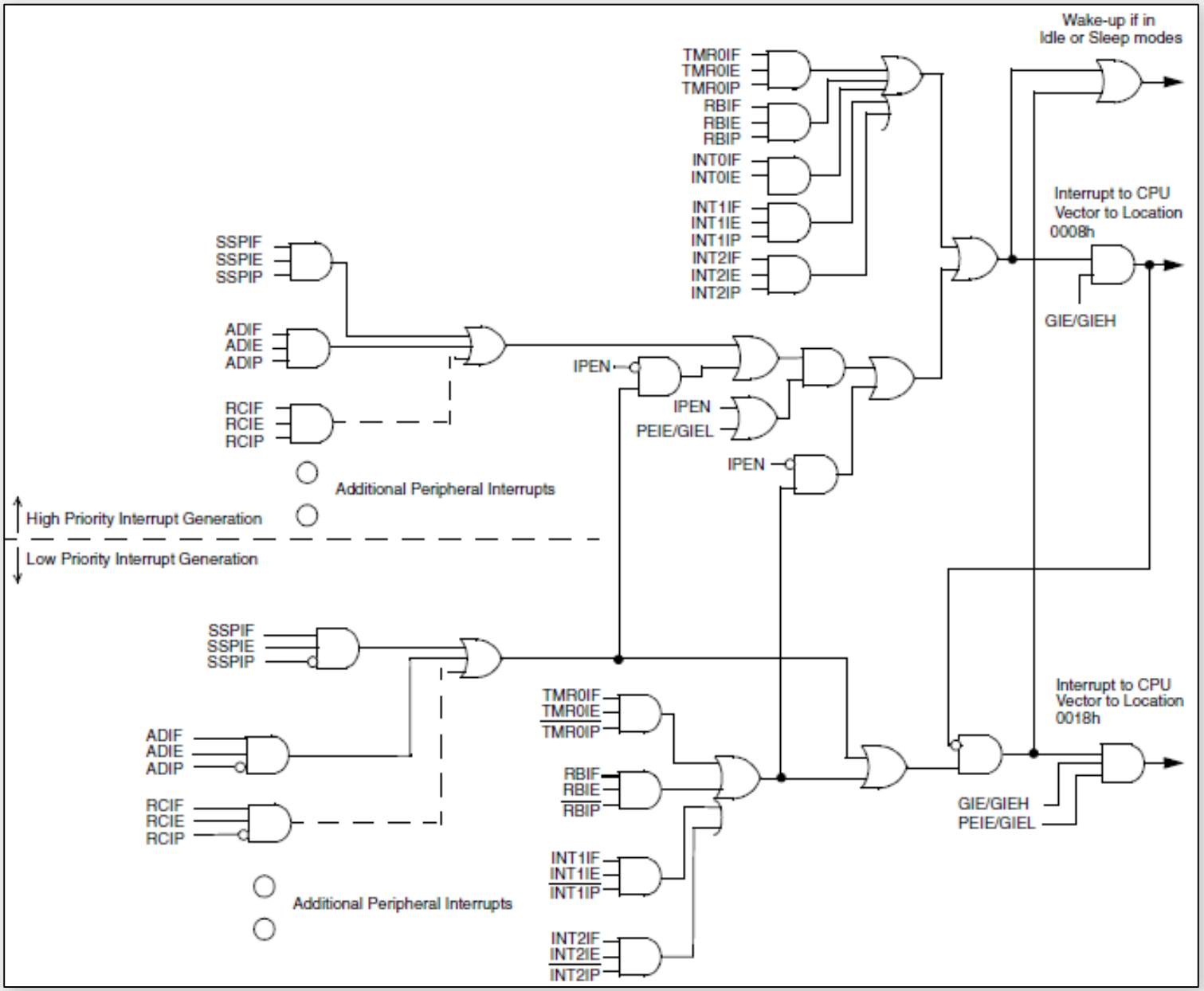
Lógica de interrupciones. Ej. µC 16F628A y 16F884

Los bits que se denominan "NNIF" (NN *interrupt flag*) corresponden al evento NN que genera el periférico, y los NNIE (NN *interrupt enable*) son los que el software pone en '1' o '0' para habilitar o deshabilitar la interrupción por ese evento.
La AND de salida es la que provoca el salto a las rutinas de servicio.
El bit GIE (*Global Interrupt Enable*) permite habilitar o deshabilitar globalmente.





Lógica de interrupciones en los µC 18F2xxx/4xxx



Registros de interrupciones en los µC 16F628A y 16F884

Registros de interrupciones en el PIC16F628A

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR Reset	Value on all other Resets(1)
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
PIR1	EEIF	CMIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF	0000 -000	0000 -000
PIE1	EEIE	CMIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE	0000 -000	0000 -000

Registros de interrupciones en el PIC16F884

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000x
PIE1	—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	-000 0000	-000 0000
PIE2	OSFIE	C2IE	C1IE	EEIE	BCLIE	ULPWUIE	—	CCP2IE	0000 00-0	0000 00-0
PIR1	—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	-000 0000	-000 0000
PIR2	OSFIF	C2IF	C1IF	EEIF	BCLIF	ULPWUIF	—	CCP2IF	0000 00-0	0000 00-0

Registros de interrupciones en los µC 18F2xxx/4xxx

NOMBRE	B7	B6	B5	B4	B3	B2	B1	B0
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
INTCON2	$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF
IPR2	OSCFIP	CMIP	—	EEIP	BCLIP	HLVDIP	TMR3IP	CCP2IP
PIR2	OSCFIF	CMIF	—	EEIF	BCLIF	HLVDIF	TMR3IF	CCP2IF
PIE2	OSCFIE	CMIE	—	EEIE	BCLIE	HLVDIE	TMR3IE	CCP2IE
IPR1	PSPIP(2)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP
PIR1	PSPIF(2)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
PIE1	PSPIE(2)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

En los microcontoladores PIC16 y PIC18 los bits de **habilitación** de interrupciones se concentran en registros **PIEx** y los **flags** de interrupciones en los registros **PIRx**. Normalmente están emparejados (Ej. RCIE es b5 de PIE1, RCIF es b5 de PIR1)

En los microcontroladores AVR, los bits de habilitación y los flags de interrupciones de cada periférico (UART, Timer etc) están en los registros de control y estado propios de dichos periféricos.

Notas:

En los µC **PIC16** todos los eventos producen finalmente el salto a una misma dirección del programa. Allí debe efectuarse el *polling* de los "nnIF" para verificar qué evento debe atenderse y a qué subrutina llamar. Obviamente en este caso, ante eventos simultáneos, el orden de verificación establece la *prioridad*.

En los PIC18 existen dos posibles direcciones de salto (0x08 o 0x18) a las cuales se pueden canalizar las **IRQ** de los distintos periféricos escribiendo su correspondiente bit en el registro IPRx. Esta diferenciación es una **forma elemental de interrupciones vectorizadas**. A su vez, las interrupciones dirigidas a 0x08 tienen prioridad por sobre las dirigidas a 0x18, es decir interrumpirán la rutina de servicio de una interrupción de menor prioridad. Esto es una forma elemental de **interrupciones anidadas**.

En los µC AVR, PIC24, dsPICs, ARMs etc las interrupciones están totalmente **vectorizadas**, esto es, cada evento produce el salto a una posición de memoria distinta, lo que reduce el *overhead* (tiempo hasta la atención del evento). En estos µC es posible establecer **prioridades**. Es posible además realizar **interrupciones anidadas** (*nested interrupts*).

En interrupciones en general hay que salvar el contexto. En el caso de anidamiento hay que cuidar además que cada rutina tenga su propio espacio de trabajo (variables estáticas).

IRQ: *Interrupt request*. Requerimiento de interrupción

Vectores de interrupción en ATmega328 and ATmega328P

Vector	Dirección	Fuente	Definición
1	0x0000	RESET	Ext Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Interrupción externa (ATmega)

EICRA – External Interrupt Control Register A

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

ISCx1-ISCx0
00 Interrumpe por nivel bajo
01 Interrumpe por cualquier cambio
10 Interrumpe por flanco de bajada
11 Interrumpe por flanco de subida

EIMSK – External Interrupt Mask Register

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

EIFR – External Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	–	–	–	–	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	

El ATmega328 tiene 2 entradas INT1 e INT0, en los pines PD2 y PD3. Se pueden configurar para interrumpir por nivel bajo, por flanco de subida, flanco de bajada o por ambos. Al ocurrir el evento que se ha configurado para el pin PD2 (PD3), se activa el flag INTF0 (INTF1) del registro **EIFR**. Si el correspondiente bit del registro EIMSK fue habilitado, es decir INT0=1 (INT1=1), y está la habilitación global (bit "**I**" del registro **SREG**) se producirá la interrupción.

Interrupción por cambio en Pin (ATmega)

PCICR – Pin Change Interrupt Control Register

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

PCIE2: Habilita *Pin Change Interrupt* en pines PCINT[23:16]
PCIE1: Habilita *Pin Change Interrupt* en pines PCINT[15:8]
PCIE0: Habilita *Pin Change Interrupt* en pines PCINT[7:0]

PCIFR – Pin Change Interrupt Flag Register

0x1B (0x3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR

PCMSK2, 1, 0 – Pin Change Mask Registers 2, 1 y 0

(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
(0x6C)	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2

El Atmega tiene entradas **por cambio** en todos sus pines de E/S, en el caso del ATMega en los puertos B, C y D.
La habilitación se realiza combinando una habilitación por grupo (Registro PCICR) y una habilitación individual de cada pin en los registros PCMSK2, PCMSK1 o PCMSK0.
Ante el cambio en cualquiera de los pines habilitados por su bit en **PCMSK**, se activa el flag PCIF2, PCIF1 o PCIF0 del registro **PCIFR**. (*Cuidado, esta interrupción activa el flag al cabo de 3 ciclos de reloj. Ver manual*)
Si el bit PCIE2, PCIE1, PCIE0 correspondiente está en ”1”, y está la habilitación global, se produce la interrupción.
Observar que en el caso de habilitar más de un pin en PCMSKx, el programa deberá evaluar cuál fue el que cambió.

Atributos de las rutinas de servicio de interrupción (ISR)

Atributo **ISR_NAKED**

En la ISR podrían modificarse **los mismos registros** que se estaban usando en la tarea interrumpida (registros R0 a R31). Es necesario entonces preservar sus valores (en la pila en RAM) antes de ejecutar la ISR, y recuperarlos al finalizar la ISR para retornar y continuar con la tarea interrumpida.

Estas operaciones de resguardo de registros (prólogo) y posterior recuperación (epílogo) – que en *assembler* se realizan con instrucciones PUSH y POP – las inserta el compilador antes y después del código que hemos puesto en la ISR, sin que debamos preocuparnos. El compilador inserta también la instrucción de retorno reti().

En ocasiones puede requerirse mayor velocidad de respuesta a un evento. Es posible decirle al compilador que no inserte nada (ya que suele preservar más registros que los necesarios)

Esto se hace con el atributo NAKED en la ISR. Por ejemplo

```
ISR(INT0_vect, ISR_NAKED)
```

Es responsabilidad del programador insertar las instrucciones que considere necesarias (PUSH y POP) para preservar y recuperar el contexto, y la inserción de la instrucción reti()

Atributo **ISR_BLOCK**

Se bloquean las demás interrupciones durante la ISR (atributo por defecto de las ISR)

Atributo **ISR_NOBLOCK**

No se bloquean las demás interrupciones durante la ISR (atributo por defecto de las ISR)

Es similar a habilitar con sei() (aunque mejor, porque se puede interrumpir durante el prólogo

Interrupciones: Ejercicios

Ejercicio 2:
Realizar un programa en ATmega328P con dos salidas en PB5 y PB2, entrada en PB3. La salida PB5 debe producir una señal cuadrada de 1 Hz permanente, y la salida en PB2 debe invertirse cuando se pulse PB3.

```
#define F_CPU 16000000
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
```

Ejercicio 3:
Realizar un programa en ATmega328P con salida en PB5, y dos entradas en PB3 y PB4.
El autómata tendrá dos estados, Activado y Desactivado. Inicialmente estará Activado, y en este estado deberá producir una señal de 1Hz, Duty Cycle 70% en PB5. Al pulsar PB4 debe pasar a modo Desactivado, y en tal estado poner la salida en 0. Al pulsar PB3 debe volver al estado Activado y generar la señal de 1Hz.

Interrupciones: Ejercicios

Ejercicio 4:
Realizar un programa en ATmega328P con tres salidas: PB5, PB2 y PB1, y 2 entradas en PB3/PCINT3 y en PD2/PCINT18/INT0.
Cada salida será manejada por una tarea distinta.
La salida PB5 será manejada en la tarea principal y oscilará permanentemente a 1 Hz.
La salida PB2 será manejada por una rutina de servicio de interrupción del tipo PCI (Pin Change Interrupt), que producirá 10 ciclos a 2 Hz.
La salida PB1 será manejada por una rutina de servicio de interrupción del tipo *External INT,* que producirá 15 ciclos a 2 Hz.
Esta interrupción deberá tendrá prioridad sobre la anterior.

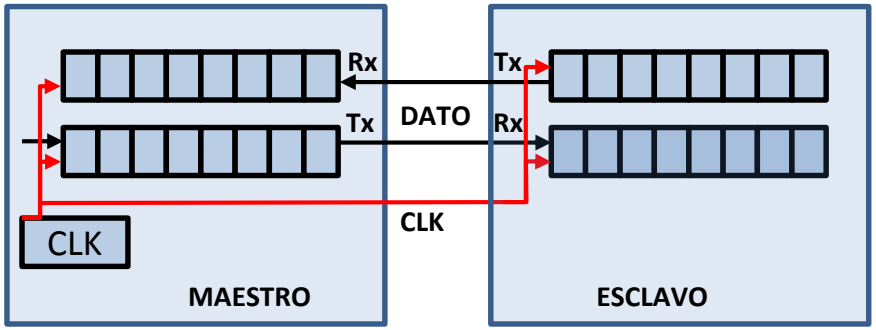
Interfaces de comunicación serie de µC

Interfaces de comunicación serie de microcontroladores

Síncronas:

Se utilizan para comunicar ICs dentro de un mismo equipo.
No utilizan adaptación eléctrica, excepto alguna R de *pull-up* (ej I2C) o para interconectar ICs de distinto voltaje de alimentación.
Muy difundidos para conectar uCs con memorias, conversores A/D-D/A e interfaces en general.

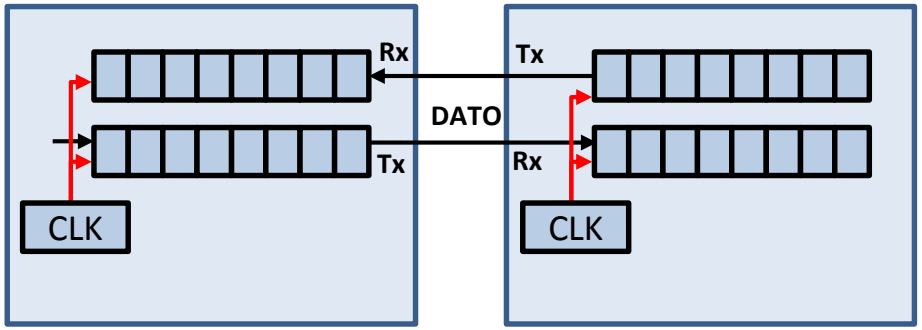
SPI *(simplex, full duplex, bus, ring)*
I2C-SMbus *(half duplex-bus)*



Asíncronas:

Para comunicar equipos a diversas distancias.
Requieren interfaz eléctrica según cada norma.

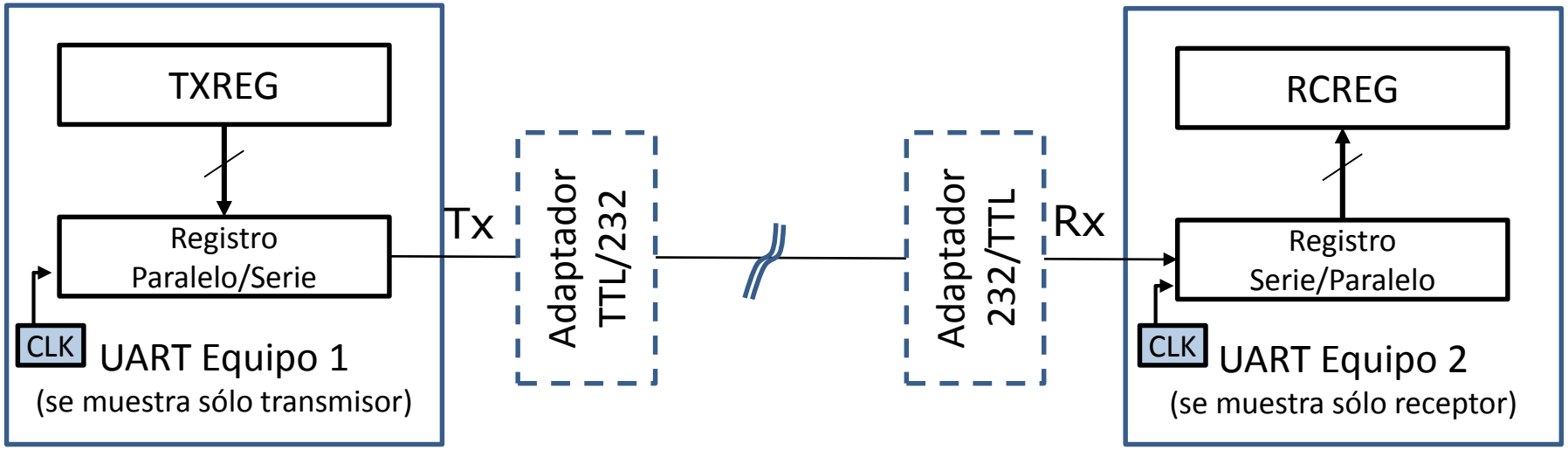
UART (232/485/422)
USB
Ethernet (incluye clk en dato)
CAN (*Controller Area Network***)**
LIN (*Local Interconnect Network***)**
IrDA (*Infrared Data Association***)**



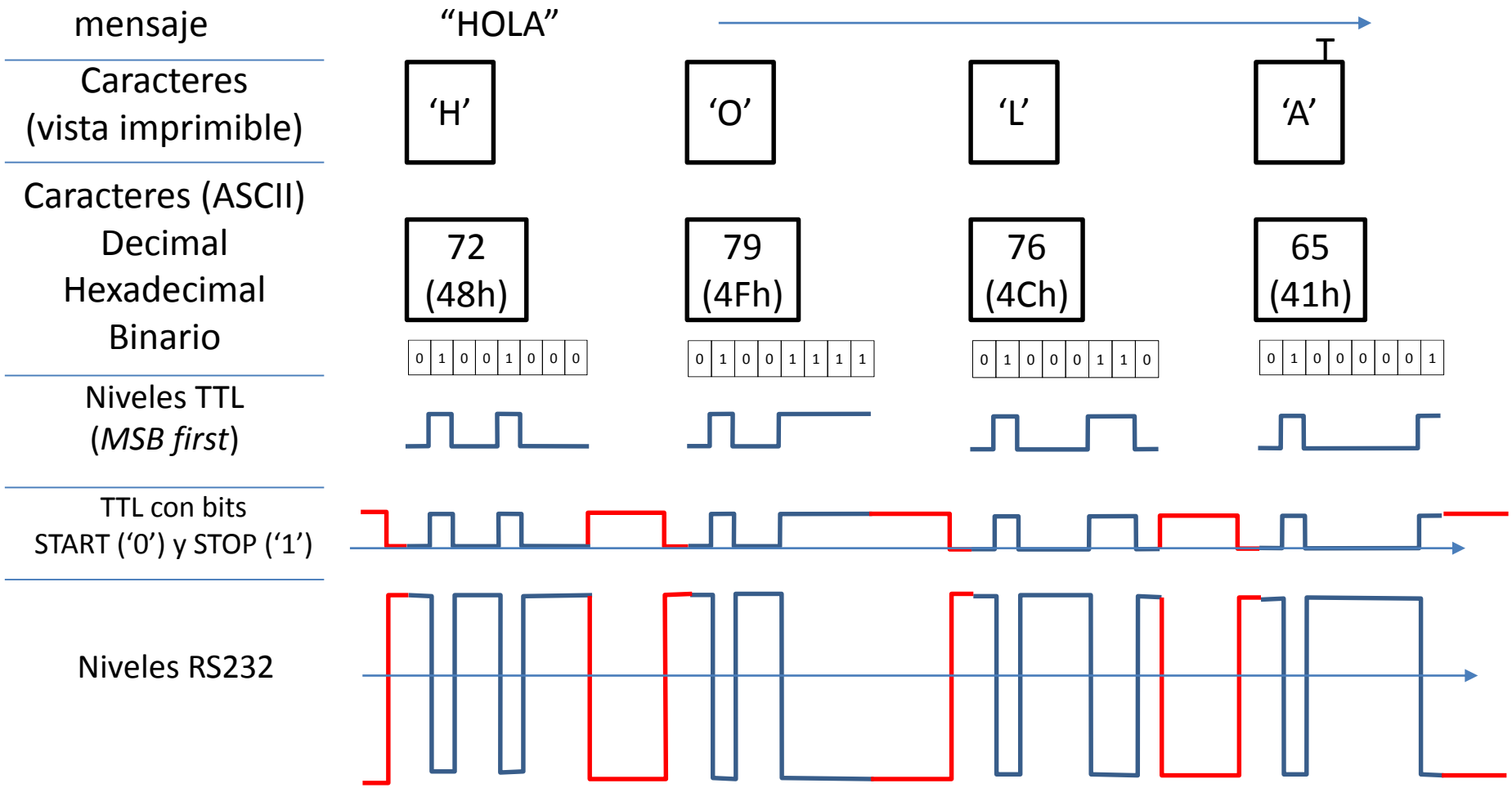
UART

UART

La UART (**Transmisor-Receptor Asíncrono Universal)** es un subsistema presente en computadoras, microcontroladores PLCs etc que se utiliza para transmitir y recibir caracteres (normalmente codificados en 1 byte denominado código ASCII). Los caracteres a transmitir por un equipo (PC, microcontrolador etc) son cargados en paralelo a un registro de la UART, denominado comúnmente *Registro de Transmisión* (TXREG en PICs, UDR Tx en AVR), y mediante un registro de desplazamiento (paralelo/serie) se presentan los bits en serie en un único pin de salida (pin Tx), como niveles altos y bajos. La salida de la UART da un nivel alto (Ej. 5 volts) para el '1' y un bajo (0 volts) para el '0', conocidos como "*niveles TTL*". Para la transmisión a distancia, en las que debe darse robustez a la señal frente a interferencias y ruido, se utilizan adaptadores eléctricos normalizados, siendo las normas más comunes RS-232, RS-422 y RS-485.
En la recepción se realiza el proceso contrario: Los bits que llegan en serie (secuencia de 'unos' y 'ceros') ingresan a un registro de desplazamiento y, una vez que han ingresado todos los bits son volcados a denominado *Registro de Recepción* (RCREG en PICs, UDR Rx en AVR), de donde puede ser leído el carácter recibido.



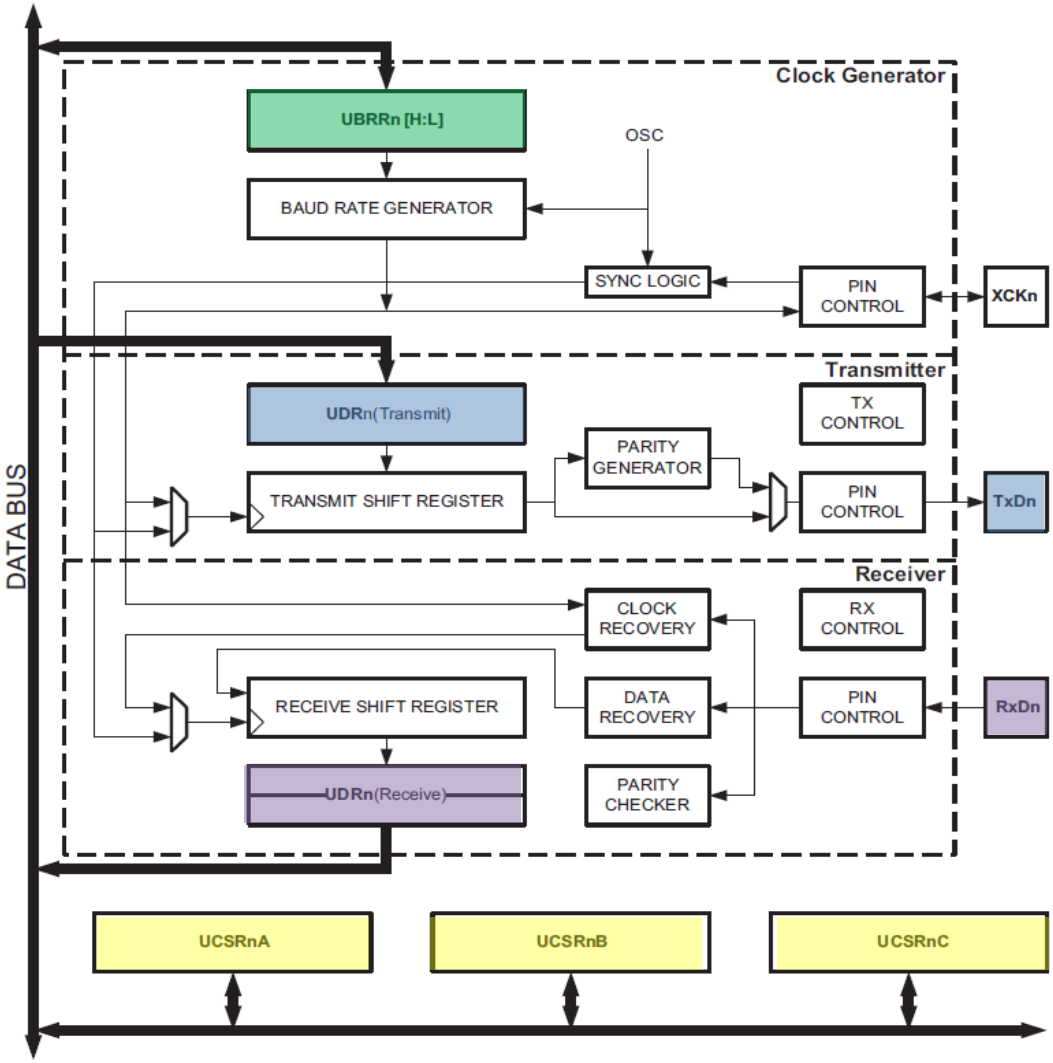
Mensaje, caracteres, bits, niveles…



Un mensaje compuesto por varios caracteres será transmitido como una sucesión de bytes, y éstos a su vez como una sucesión de bits. En transmisión asíncrona se **delimitan** los caracteres mediante bits adicionales: un bit de START ('0') y entre uno y dos bits de STOP ('1'). Entre caracteres habrá entonces un STOP y luego un START, garantizando así un flanco de sincronismo. Luego del último carácter la línea queda en "silencio", que es un '1'. Cuando comience un nuevo mensaje, el primer carácter viene con su START en '0', produciendo el flanco de bajada que permite reconocer el instante en que comienza.

USART del ATmegaxx

El ATmegaxx trae una interfaz USART (Universal Synchronous Asynchrnous Receiver Transmiter), esto es, puede trabajar como en modo asíncrono o síncrono (con pin de clock). Nos centraremos en el modo asíncrono o modo UART



Registros asociados a la USART del ATmegaxx

UDRn **USART Data register**

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDRn (Read)
	TXB[7:0]								UDRn (Write)
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	

Son **los registros** de Datos recibidos/a transmitir. Comparten la misma dirección, pero **cuando se lee** se accede al valor en el *buffer de* ***recepción*** RXB, y **cuando se escribe** el valor pasa al *buffer de* ***transmisión*** TXB.

UBRRn **USART Baud Rate register**

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRRn[11:8]				UBRRnH
	UBRRn[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	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	
	0	0	0	0	0	0	0	0	

Este par de registros contiene el factor de division del clock. Dado un valor deseado de BaudRate debe ser **UBRRn = F_CPU/16/Baudrate-1**. O dado el UBRRn, será **BaudRate = F_CPU/16/(UBRRn+1)**
Ejemplo: F_CPU = 16MHz, BaudRate 9600bps → UBRRn = 16000000/16/9600-1 = 103,16 = 103
Esto da un BaudRate = 16000000/16/(103+1) = 9615,38. Error: 0,16%.

Nota: Si el bit **U2xn** del registro **UCSRnA** (ver siguiente página) se pone en '1' la velocidad se duplica

Registros asociados a la USART del Atmegaxx (2)

UCSRnA **USART Control and Status Register A**

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

RXCn: flag de recepción completa. Se borra al leer UDRn. Puede usarse para provocar interrupción.
TXCn: flag de transmisión completa. Puede usarse para provocar interrupción.
UDREn: flag de UDRn (transmisión) vacío. Puede usarse para provocar interrupción.
FEn: flag de ***error de frame*** (se activa cuando el bit de stop es 0).
DORn: flag de ***error de overrun***. (se activa cuando buffers de recepción y UDRn están completos y llega un nuevo bit de start.
UPEn: flag de ***error de paridad***.
U2Xn: bit de control para duplicar el *baudrate*. Con 1 duplica, y será **BaudRate = F_CPU/8/(UBRRn+1)**
MPCMn: bit de control para habilitar modo multiprocesador (maestro-esclavo).

UCSRnB **USART Control and Status Register B**

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

RXCIEn: Habilita interrupción por RXCn.
TXCIEn: Habilita interrupción por TXCn.
UDRIEn: Habilita interrupción por UDREn.
RXEn: Con '1' habilita la recepción
TXEn: Con '1' habilita la transmisión
UCSZn2, junto con los bits UCSZn1:0 (del registro UCSRnC) determina la longitud del dato (5 a 9).
RXB8n y **TXB8n** son los novenos bits de Rx y Tx en caso de usar 9 bits de dato.

Registros asociados a la USART del Atmegaxx (3)

UCSRnC **USART Control and Status Register C**

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
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	1	1	0	

UMSELn1:0 Seleccionan el modo de trabajo entre 00:USART asíncrona 01: USART síncrona, 11: SPI master
UPMn1:0 Seleccionan entre 00: no paridad, 10: paridad Par, 11: paridad Impar
USBSn: Define la cantidad de bits de stop a insertar. 0: 1 bit de stop, 1: 2 bits de stop
UCSZn1:0 Junto con el bit UCSZn2 visto antes definen: 000:5 bits, 001:6 bits, 010:7 bits, 011:8 bits, 111:9 bits
UCPOLn: Solo para modo síncrono. Determina polaridad entre dato y clock.

En el RESET queda configurado como USART asíncrona (UART), 8 bits, 1 bit de STOP

Funciones de UART en Atmel Studio

Para implementar la transmisión y recepción de bytes a través de la UART se deben escribir 3 funciones: de inicialización, de transmisión y de recepción.

```
//        Inicialización
void mi_UART_Init( unsigned int brate)
{
UBRR0 = F_CPU/16/brate-1;// Configura baudrate. Ver en sección UART de datasheet
UCSR0A = 0; // Asegura que el bit U2x0=0 (velocidad normal)
UCSR0B = (1<<RXEN0)|(1<<TXEN0);// Habilita bits TXEN0 y RXEN0
UCSR0C = (1<<USBS0)|(3<<UCSZ00);// USBS0=1 2 bits stop, UCSZxx=3 8 bits
}


//        Transmisión
int mi_putc(char c)
{
while(!(UCSR0A & (1<<UDRE0)) ); // Espera mientras UDRE0=0 (transmisión ocupada)
UDR0 = c;// Cuando se desocupa, UDR0 puede recibir el nuevo dato c a trasmitir
return 0;
}

//        Recepción
int mi_getc()
{
while ( !(UCSR0A & (1<<RXC0)) );//Espera mientras el bit RXC0=0 (recepción incompleta)
return UDR0;                    //Cuando se completa, se lee UDR0
}
```

La UART como E/S estándar en Atmel

Para utilizar las funciones de la biblioteca *stdio.h* (E/S estándar) como *printf*, *scanf* etc es posible redefinir las funciones *putc* y *getc* (ó *fputc* y *fgetc*) con nuestras funciones de E/S de UART (también de otras interfaces). De esta forma será posible por ejemplo transmitir números formateados en decimal, hexadecimal, punto flotante etc.

Al inicio del programa debe declararse y definirse lo siguiente:

```
#include <stdio.h>// para las funciones estándar de E/S

void mi_UART_Init( unsigned int);// Idem página anterior
int mi_putc(char, FILE *stream);//Con un segundo argumento tipo FILE para compatibilidad
int mi_getc(FILE *stream);

FILE uart_io = FDEV_SETUP_STREAM(mi_putc, mi_getc, _FDEV_SETUP_RW); // Inicializa un
tipo stream de E/S con las funciones de salida y entrada

#define fgetc() mi_getc(&uart_io)   // redefine la primitiva de entrada
#define fputc(x) mi_putc(x,&uart_io)// redefine la primitiva de salida
```

En el main debe apuntarse el stream uart_io como E/S estándar

```
stdout = stdin = &uart_io;
```

Nota: Estas funciones están descriptas en el manual de AVR Libc.
https://www.nongnu.org/avr-libc/

Múltiples UARTs como streams de E/S (Atmel)

De manera similar a la función printf, es posible utilizar la función fprintf para transmitir por otros dispositivos, por ejemplo por la UART1 en ATmega2560 (que tiene 4 UARTs)

1- Al principio del programa, o en un archivo cabecera (Ej misUARTs.h)

```
/*UART 0 (stdio)   */
void UART0_Init( unsigned int);
int putchar0(char c, FILE *stream);
int getchar0(FILE *stream);
/*UART 1    */
void UART1_Init( unsigned int);
int putchar1(char c, FILE *stream);
int getchar1(FILE *stream);
```

Estas funciones implementadas para cada UART

```
FILE uart0_io = FDEV_SETUP_STREAM(putchar0, getchar0, _FDEV_SETUP_RW);
FILE uart1_io = FDEV_SETUP_STREAM(putchar1, getchar1, _FDEV_SETUP_RW);
```

2- En el main elegir la UART que va a ser la E/S estándar (puede ser cualquiera)

```
stdout = stdin = &uart0_io;
```

3- Utilizar *printf* para la E/S estándar, *fprintf* para las otras UARTs como *streams*

```
printf("Pos:%d\r\n",posicion);
fprintf(&uart1_io,"Pos:%d\r\n",posicion);
```

El primer argumento apunta al *stream* al que se dirige *fprintf*

UART y sprintf

De una forma más portable que la anterior (propia de avrlibc) puede utilizarse la función sprintf para formatear datos a cadenas. Luego implementando una función similar a puts(), se canaliza la cadena al dispositivo deseado (en este caso una UART)

1- Al principio del programa, o en un archivo cabecera (Ej misUARTs.h)

```
void UART0_Init(unsigned int);   // Función que inicializa la UART
int putchar0(char);              // Función para transmitir caracter por UART
int getchar0 ();                 // Función para recibir caracter por UART
int puts0(char *cadena);         // Función para canalizar sprintf a putchar0
char Tx_stream[30];              // Array para recibir cadena desde sprintf
```

2- Utilizar s*printf hacia la cadena, y* la función puts0 para enviar cada carácter la función putchar0()

```
sprintf(Tx_stream,"Pos:%d\r\n",posicion);
puts0(Tx_stream);
```

```
int puts0(char *cadena){
int n=0;
while (*cadena!=0){mi_putc(*cadena++);n++;}
return n;}
```

Interrupciones de UART

Interrupción de Recepción
Es posible verificar periódicamente el flag de recepción **RXCn** para comprobar si hay nuevo carácter para leer con getc(). Si se recibe un segundo carácter antes de leer **UDRn** bajará al *buffer* FIFO, y si se recibe un tercero quedará en el Registro de Desplazamiento y no pasará a **UDRn**. Si sucede el bit de START de un **cuarto** carácter antes de leer **UDRn**, se produce un error de ***overrun*** (que puede detectarse leyendo el bit **DORn**). Debe evitarse la situación de ***overrun*** porque implica pérdida de datos y generalmente pérdida de tramas completas. Los µCs más potentes suelen traer una UART con un *buffer* FIFO más grande, e incluso se puede programar un DMA (acceso directo a memoria) para volcar los caracteres directamente en un área de RAM sin intervención de la CPU, a medida que van llegando.
Un modo de evitar el ***overrun*** es habilitar la interrupción por recepción de dato serie **RXCIEn** .

Interrupción de Transmisión
Transmitir **un** carácter no le insume prácticamente tiempo al µC, pues escribe el dato en **UDRn** (en apenas un ciclo de reloj) y puede continuar con otras tareas. El carácter pasa al Registro de Desplazamiento de Transmisión y de allí se transmitirá en un tiempo que depende del Baudrate. Por ejemplo, a 9600 bps, un carácter (8+2=10 bits) tarda algo más de 1ms en ser transmitido. Un segundo carácter puede ser puesto en **UDRn** mientras el anterior se transmite. No puede ponerse un tercer carácter hasta tanto no se haya transmitido el primero. Así, un µC perderá varios milisegundos si quiere transmitir un mensaje largo. Se puede utilizar la interrupción de transmisión (**TXCIEn**=1) que avisa cuándo se puede escribir **UDRn**, y así evitar las esperas.

Interrupciones de UART en ATmega

Como se vio en los registros asociados a la UART, el registro UCSRnA contiene los *flags* de interrupción por recepción (RXCn) y transmisión (TXCn, UDREn) y el registro UCSRnB los correspondientes bits habilitadores (RXCIEn, TXCIEn, UDRIEn).

UCSRnA **USART Control and Status Register A**

RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn

UCSRnB **USART Control and Status Register B**

RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n

De manera similar a como se hizo para las interrupciones externas, se puede escribir las rutinas de servicio de las interrupciones de Transmisión o Recepción.
Una rutina de servicio de la interrupción por recepción de dato en la UART tendría la forma:

```
ISR(USART_RX_vect)
{ char Dato;
 Dato=getchar(&uart_io);
 switch(Dato){
…}
}
```

incluso es posible directamente
`Dato = UDR0;`
ya que no es necesario el polling del bit ***RXCn***

Intérprete de comandos

Una forma práctica de configurar, depurar y supervisar un programa en microcontrolador es mediante comandos desde una aplicación o un terminal.

Cada comando con sus parámetros tendrá un formato similar a una trama de comunicaciones. Este formato podría respetar algún estándar (Ej. MODBUS) o ser un "protocolo" propio.

- Por ejemplo, en un protocolo propio, un comando para configurar el período en un generador de pulsos en 300 milisegundos podría ser:

:T305\r con delimitador de inicio ":", comando "T", parámetro numérico "305" en cadena decimal, y delimitador de final "\r" (CR, o ASCII 13). nota: El parámetro numérico podría ser de logitud fija, por ejemplo con longitud fija de 4 dígitos, el anterior quedaría **:T0305\r**

- Un comando para pedir el número de pulsos generados podría ser:

:N\r sin parámetros numéricos, solamente requiere respuesta

- Un comando para poner en cero el contador de pulsos podría ser:

:R\r sin parámetros numéricos, solamente ejecuta acción.

Estas tramas podrán ser interpretadas por el microcontrolador una vez que se recibe el delimitador de final.

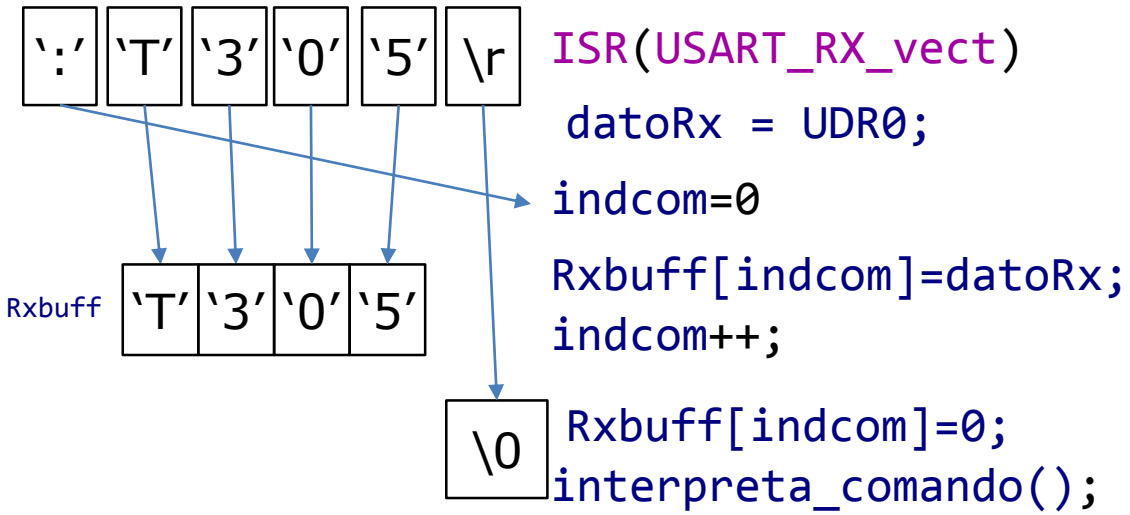
La técnica recomendable es entonces utilizar un *array* en el que se van copiando los caracteres de la trama a medida que se reciben, en la ISR de Recepción por UART.

Intérprete de comandos (2)

Utilizaremos dos variables globales: un array estático `Rxbuff[]` y un uint8_t `indcom`

```
#define RXLEN 20        // longitud máxima
char Rxbuff[RXLEN];     // array de recepción de comando
uint8_t indcom=0;       // índice
```

Caracteres recibidos (ejemplo: :T305\r)




```
ISR(USART_RX_vect)
{
char datoRx;
datoRx=UDR0;
switch(datoRx)
{
case ':':  indcom=0;
           break;
case '\r': Rxbuff[indcom]=0;
           interpreta_comando();
           break;
default:   Rxbuff[indcom]=datoRx;
           indcom++;
           break;
}
}
```

```
void interpreta_comando(){
switch(Rxbuff[0]){
case 'N':  sprintf(Txbuff,"N=%d\r\n",numpulsos);
           mi_puts(Txbuff);
           break;
case 'T':  tiempo=atol(&Rxbuff[1]); //
           break;
...
}
```

Nota: organización modular del código

Cuando un programa implementa muchas tareas distintas o interactúa con distintos periféricos conviene dividirlo en módulos (archivos separados).

Adicionalmente, si estos módulos están bien **delimitados**, pueden ser **reutilizados**.

Por ejemplo un módulo que atiende todas las funciones de comunicación serie por UART puede reutilizarse en distintos programas.

A estos módulos para **atender periféricos** podemos denominarlos ***drivers***.

Otros módulos podrían ser bibliotecas de funciones de procesamiento etc.

Driver de UART como E/S estándar

1. En **main.c** poner

```
#include "main.h“
la función main() y otras funciones de usuario
```

2. En **main.h** poner todas las declaraciones de variables, de constantes, de bibliotecas de uso global, de funciones implementadas en main.c y de funciones “drivers” de periféricos (en este caso UART)

```
#define F_CPU 16000000
#define BAUD 250000
#include <avr/interrupt.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay.h>
#include <avr/io.h

#include "UART.h"  // funciones de UART del usuario (nuestro “driver” de UART)
```

Driver de UART como E/S estándar (2)

3. En **UART.h** poner las declaraciones de funciones específicas de periférico

```
void mi_UART_Init( uint32_t, uint8_t IntRX, uint8_t intTX);
int mi_putchar(char, FILE *stream);
int mi_getchar(FILE *stream);
```

4. EN **UART.c** incluir cabeceras de main y UART, declaración de tipo FILE para el inicializador de stream stdio provisto por el usuario, inicializado con *FDEV_SETUP_STREAM* (de avr-libc)

```
#include "main.h"
#include "UART.h"
FILE uart_io = FDEV_SETUP_STREAM(mi_putchar, mi_getchar,
_FDEV_SETUP_RW);
```

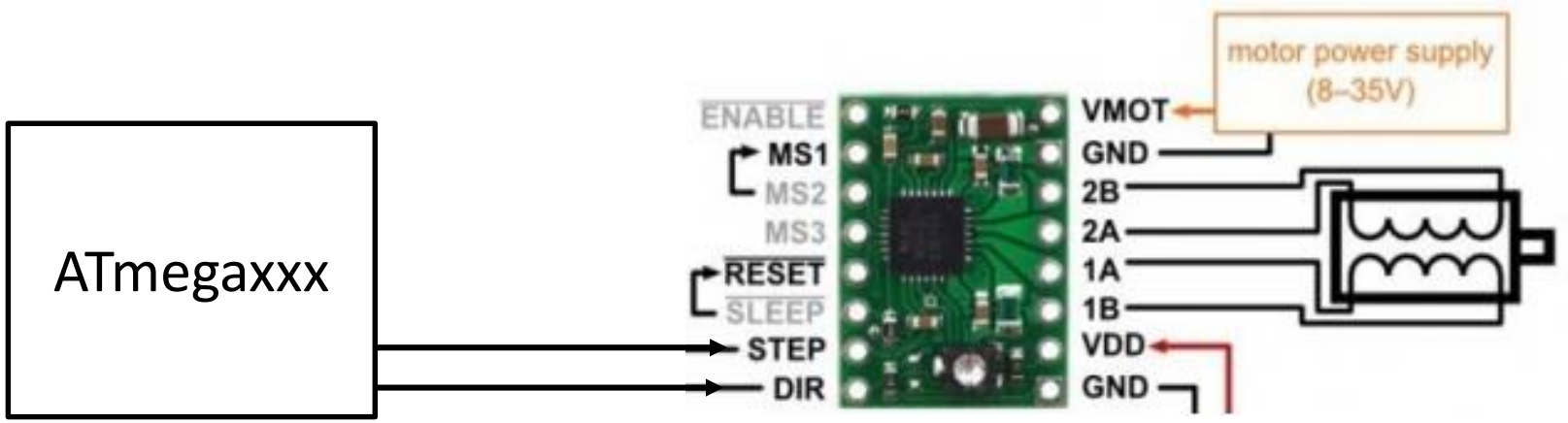
y la implementación de funciones declaradas en UART.h, entre ellas

```
void mi_UART_Init( uint32_t brate, uint8_t IntRX, uint8_t IntTX)
{
        stdout = stdin = &uart_io;
...
```

Ejercicios propuestos:

1 Realizar un oscilador con semiperíodo ajustable por consignas por UART de tipo ":Txx", con xx tiempo en décimas de segundo de 1 a 99 ((0,1 9,9 segundos). La recepción de datos no debe detener el funcionamiento del oscilador.
while(1) { S=1;_delay_ms(T);S=0;_delay_ms(T);} //S pin de salida

2 Realizar el siguiente automatismo supervisado:
Control Pulso-Dirección de un driver de motor PaP o servomotor, mediante 2 pines de PORTB.
Recibe consigna de posición P(0 a 30000 pulsos) y de período T(100 a 30000 µs) como cadenas decimales.
Debe aceptar modificación de la consigna durante el movimiento.



Temporización

Control del tiempo

En muchas aplicaciones se requiere un control más o menos preciso del tiempo. Puede ser para **generar** intervalos de tiempo entre acciones o para **medir** el tiempo transcurrido entre eventos.

- Generar pulsos para el parpadeo de un led.
- 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 RC, o para control de potencia por PWM.
- Muestrear señales provenientes de sensores. (Ej: Acelerómetros)
- Realización 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 US etc).

A las acción de “generar” o medir tiempos se las denomina **temporización**.

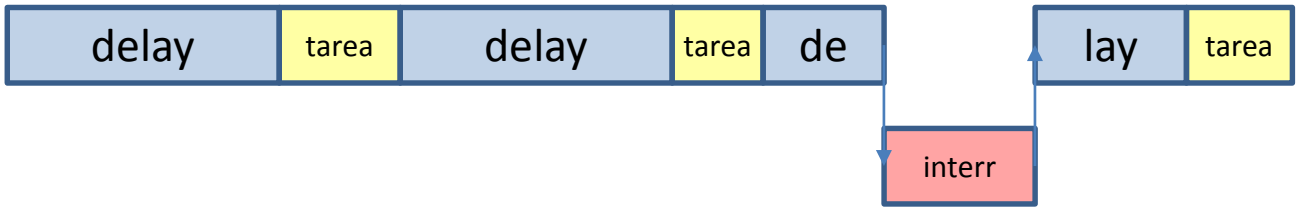
Temporización mediante funciones de retardo

La forma más sencilla de generar retardos entre acciones es mediante funciones de ***delay***, que están implementadas como ciclos de operaciones neutras (con el único propósito de consumir ciclos).

Los compiladores disponen para este fin de las funciones de delay.
Por ejemplo, en CCS PICC se dispone de delay_ms(milisegundos), delay_us(microsegundos), donde el argumento puede ser **constante** o **variable** de **16 bits**, y delay_cycles(ciclos), donde el argumento puede ser **constante** de **8 bits**.
En avr-libc (biblioteca para micros Atmel) 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 acumula el tiempo de atención de dicha interrupción.
- Se acumula el tiempo de ejecución de otras tareas (muestreo, lectura/escritura de puertos, saltos, incrementos de variables etc).



Módulo Temporizador o *timer* en microcontroladores

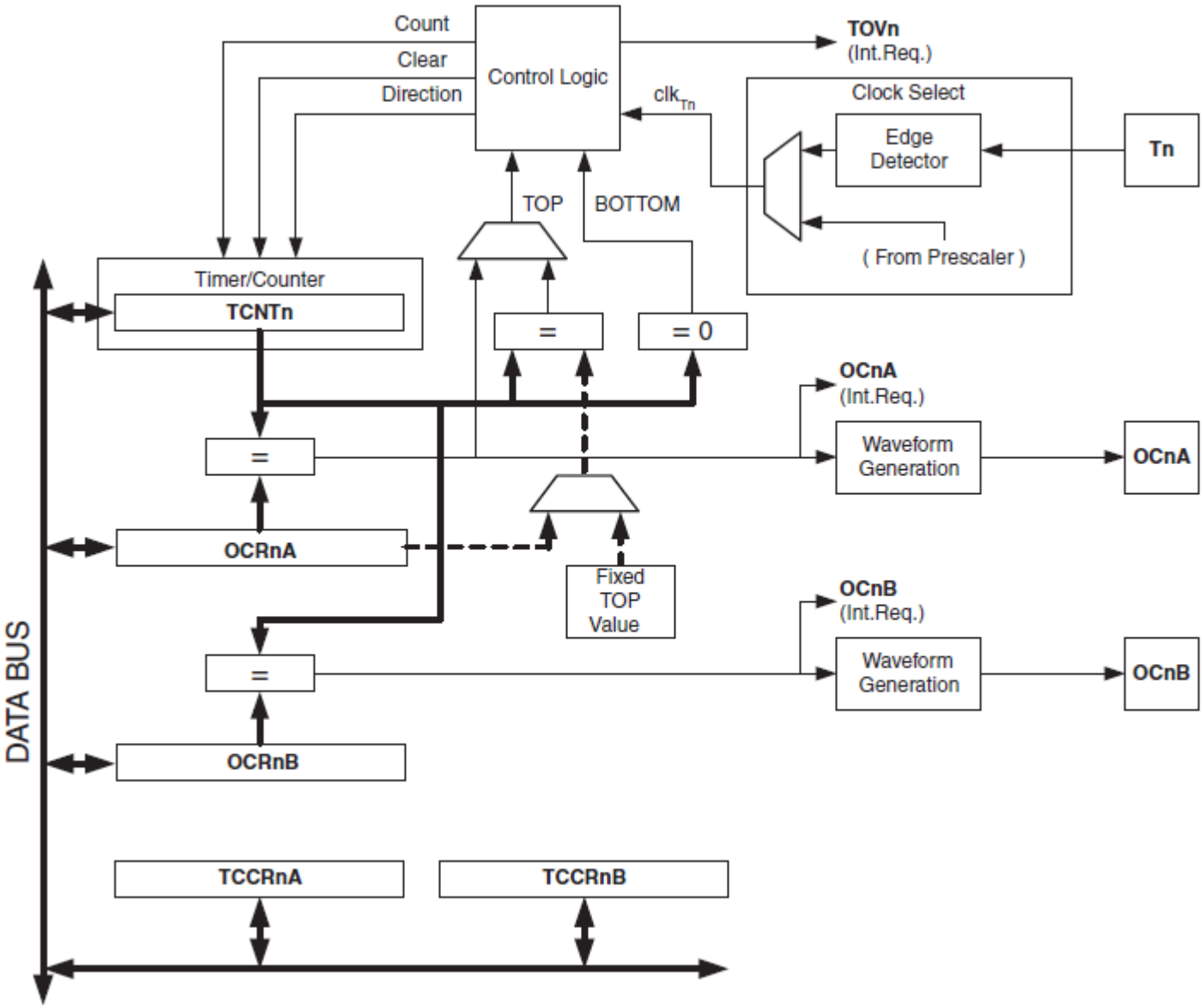
La forma más correcta de realizar un control preciso de tiempo es mediante el uso de temporizadores o *timers*.
Un microcontrolador básico dispone de al menos un *timer*, habitualmente varios, desde 8 bits hasta 32 bits.
Un *timer* es básicamente un contador que se puede leer o escribir, cuya entrada de reloj puede conectarse a distintas fuentes de pulsos.
Cuando la fuente de pulsos son eventos en un pin de entrada funciona como un simple **contador**, por ejemplo para contar pulsos de un *encoder* incremental.
Cuando los pulsos provienen de un reloj de F conocida, normalmente el clock interno, funciona como *timer*, porque provee referencia temporal.

Los timers cuentan con circuitos accesorios para:

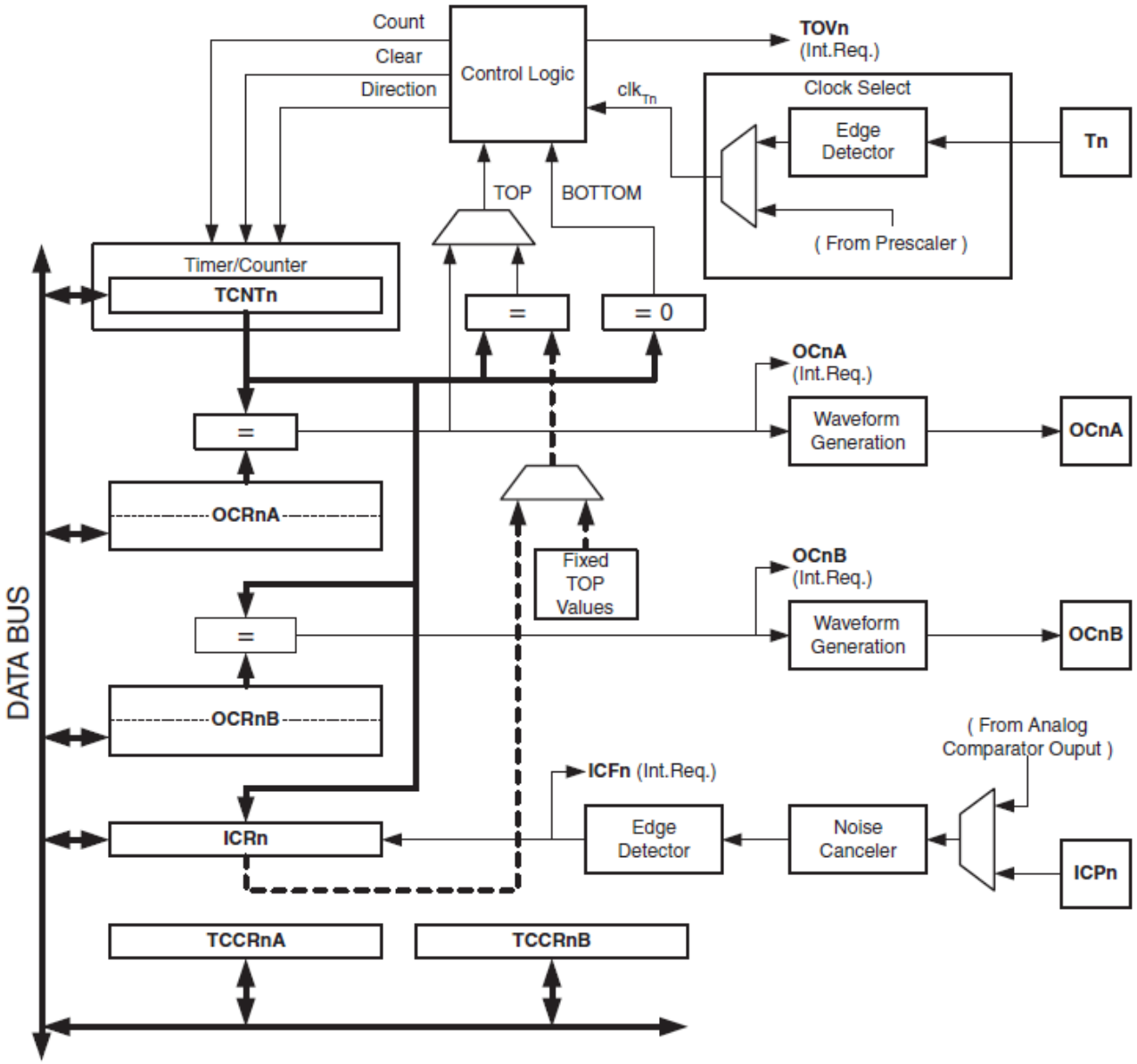
- Generar pulsos de duración, frecuencia, fase o duty cycle determinados, en modos denominados *Output Compare* y *PWM*, en una o más salidas que utilicen la misma base de tiempo.
- Cronometrar eventos en una o más entradas (tiempo entre flancos de subida/bajada, flancos sucesivos etc). Modo denominado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

Este conjunto se denomina Módulo Capture-Compare-PWM (CCP)

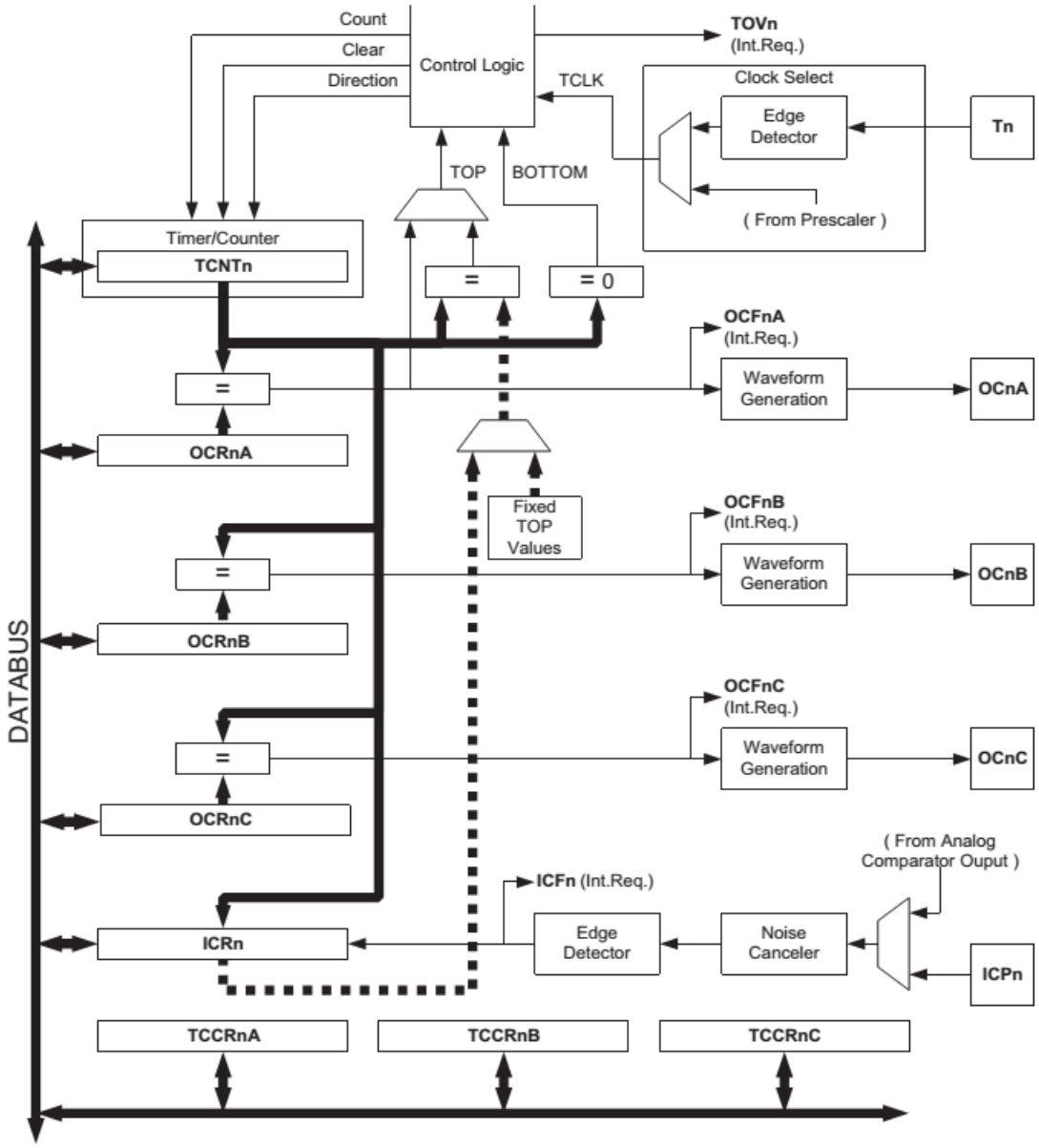
Temporizador o *timer* de 8 bits en ATmegaXX



timer de 16 bits en ATmegaXX

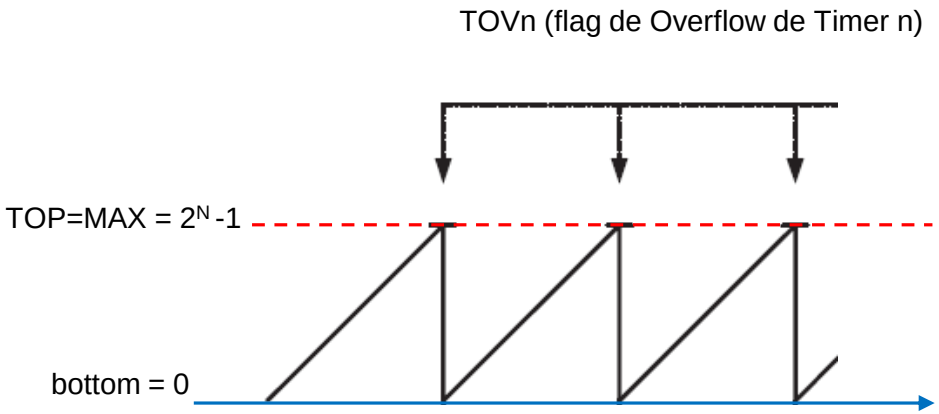


timer de 16 bits en ATmega2560, ATmega32u4 etc



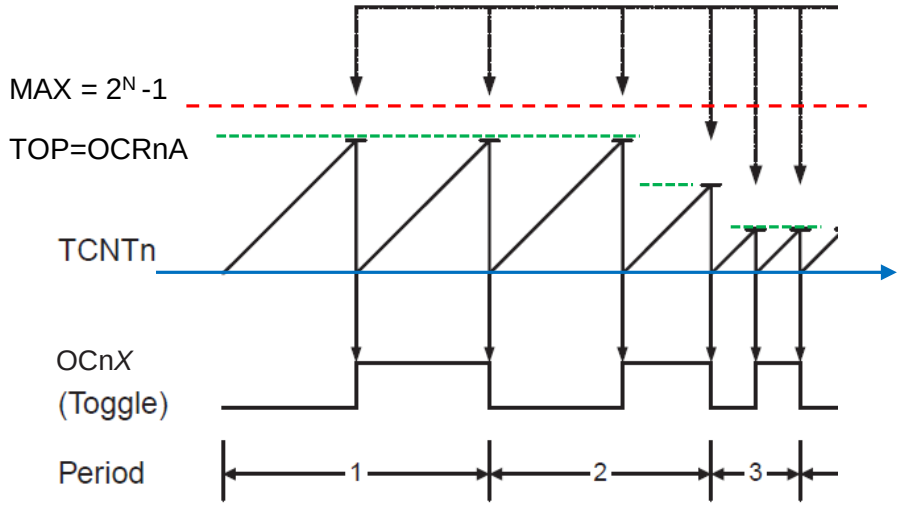
Modos de trabajo de un timer en AtmegaXX

Modo Normal (WGMn =0)
El Timer n cuenta desde 0 y cuando llega al valor MAX (255 o 65535) desborda y vuelve a 0. Al volver a 0 pone a ‘1’ el bit TOVn. (flag de *overflow*) que puede usarse para interrumpir



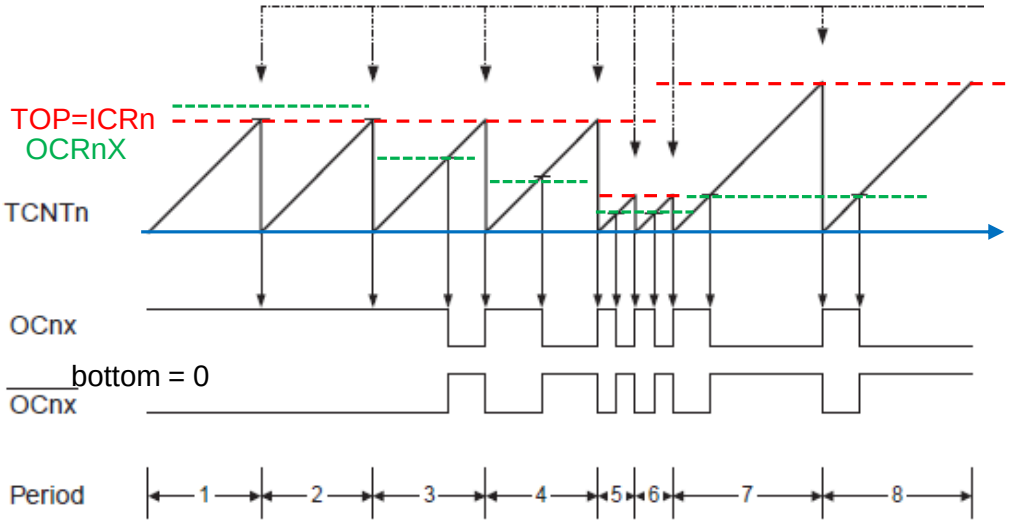
Modo *Clear Timer on Compare match* (CTC). (WGMn =4)
El Timer **n** cuenta desde 0 y cuando llega al valor de OCRn*X* se resetea y pone a ‘1’ el flag OCn*X* *(X es A ó B).* Puede también actuar sobre el pin OCnX.

En Timer de 16 bits, además de OCRnX también puede usarse el registro ICRnX (WGM1=12)

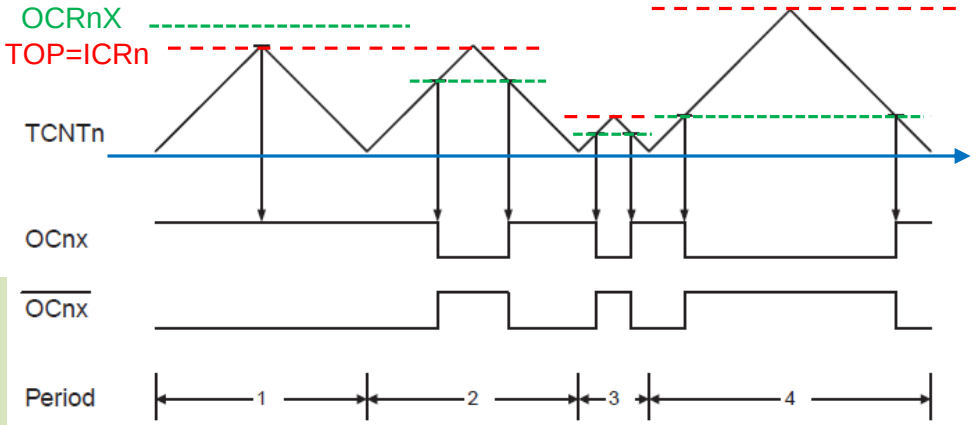


Modos de trabajo de un timer en AtmegaXX (2)

Modo Fast PWM (WGM= 14)
El Timer n cuenta desde 0 y cuando llega al valor dado por ICRn vuelve a 0.
La salida OCnX se pone en ‘1’ al comiezo del período, y baja a ‘0’ cuando el Timer n iguala al OCRn*X* .



Modo *Correct Phase PWM* (WGM=8)
El Timer **n** cuenta ascendente-descendente entre 0 y TOP, conmutando OCnX *(X es A ó B)* cuando el Timer **n** iguala a OCRn*X.*



Hay otros modos, estos son los más versátiles porque permiten modificar **Período** con **ICRn** y **Duty Cycle** con **OCRn**. El registro **ICRn** (y por tanto estos modos) sólo está en los timers de 16 bits. Si se requiere Input Capture debe usarse otro modo con TOP fijo, por ejemplo WGM=3

Registros de Control de timer en AtmegaXX (2)

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	

Comportamiento de Salida OC1A

Comportamiento de Salida OC1B

00 desconectado
01 Toggle
10 Clear
11 Set (inverso a 10)

Prescaler 0(des),1,8,64,256,1024, ext, ext

Modo de generación de Onda

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

Registros de Control de timer en AtmegaXX (2)

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								TCNT1H
(0x84)	TCNT1[7:0]								TCNT1L
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	

Bit	7	6	5	4	3	2	1	0	
(0x89)	OCR1A[15:8]								OCR1AH
(0x88)	OCR1A[7:0]								OCR1AL
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	

Bit	7	6	5	4	3	2	1	0	
(0x8B)	OCR1B[15:8]								OCR1BH
(0x8A)	OCR1B[7:0]								OCR1BL
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	

Bit	7	6	5	4	3	2	1	0	
(0x87)	ICR1[15:8]								ICR1H
(0x86)	ICR1[7:0]								ICR1L
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	

Timer

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Habilita interrupciones por Captura de entrada, Match de OCR1X y Overflow

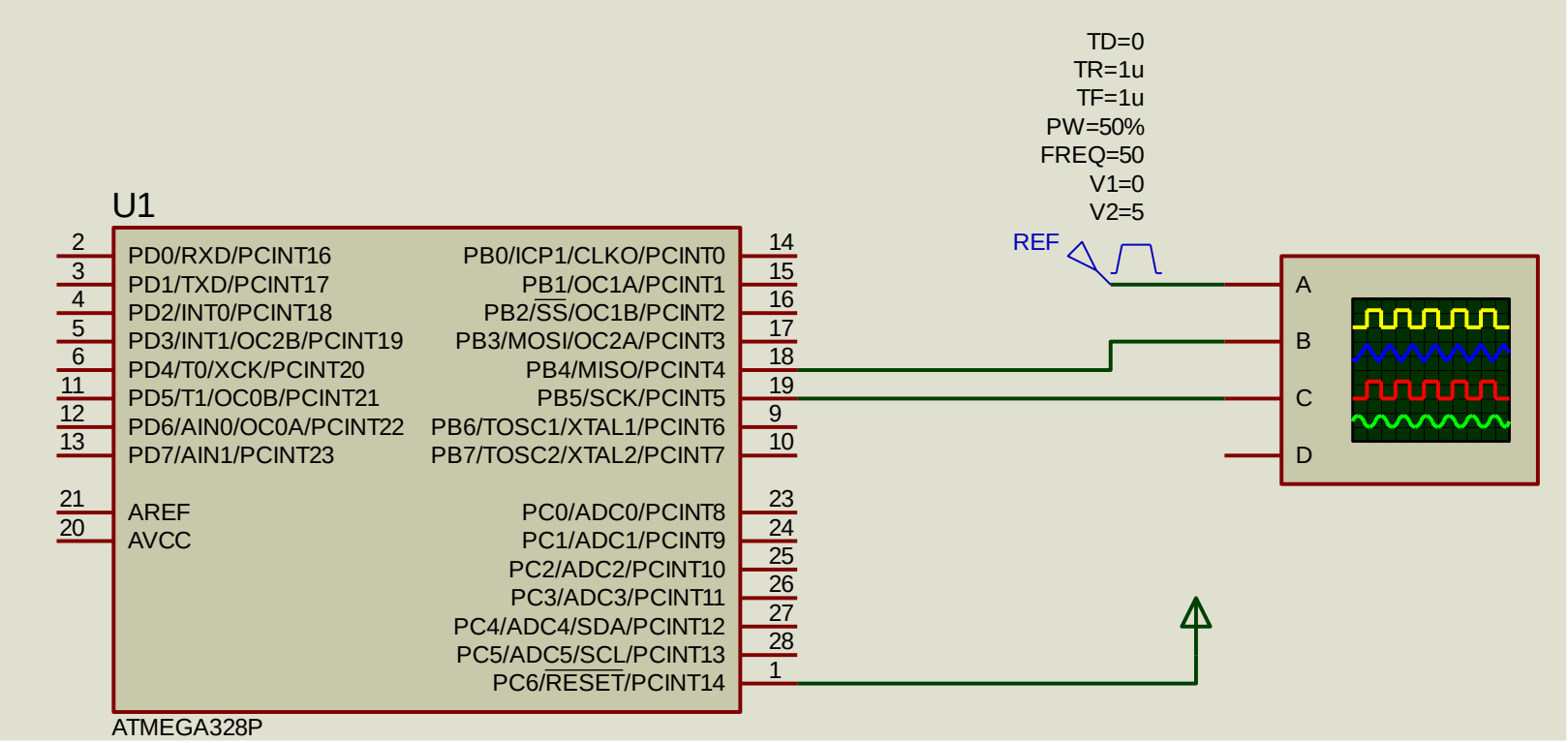
Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Flags de Captura de entrada, Match de OCR1X y Overflow

Ejemplo: Temporización: Delay vs Timers

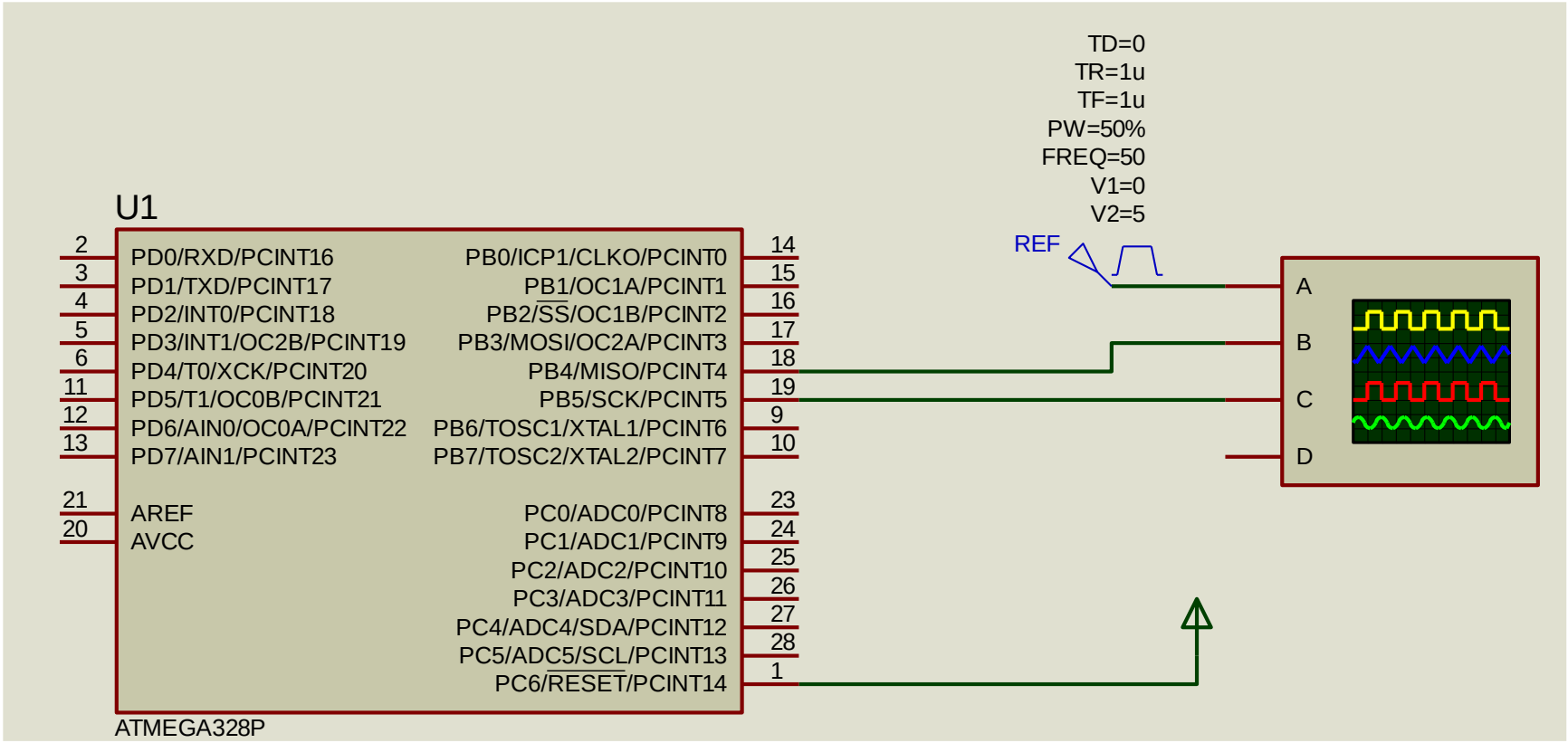
Programa mínimo de "*blink*" (activar/desactivar pines de salida): RB4 mediante delay, y RB5 mediante interrupción de Timer. En osciloscopio se comparará la señal generada con una señal de referencia de período preciso (50Hz).

Circuito Ej01_Timer_ATmega328.DSN, programa EJ01_Timer_INT.c



Ejemplo: Ensayo de distintos modos de PWM en ATmegaxxx

Circuito EJ03_PWM.DSN, programa EJ03_PWM.DSN



Muestra el uso de TIMER1 y distintos modos de generación de ondas, modificables de forma interactiva a través de comandos por puerto serie.

:Pn con n 1 a 5, cambia prescaler de 1 a 1024 (1,8,64,256,1024). 0 desactiva el timer. 6 y 7 son clk externo

:Tnnnn con n de 2 a 65536 cambia período de onda (en modos 8, 10, 14) modificando ICR1.

:DAnnnn o :**DBnnnn** establece el Duty Cycle de OC1A y OC1B modificando OCR1A u OCR1B (con nnnnn<T)

:Mn con n de 0 a 15 cambia modo de generación (ver tabla 16.4 en manual Atmega328p)

:CAn o **:CBn** con n=0,1,2,3 cambia comportamiento de las salidas OC1A y OC1B

Ejercicios propuestos:

EJ01: Realizar Blink de un led de 1 segundo con Timer 1, en una ISR.

EJ02: Reemplazar el delay del programa de control de motor PaP por temporización con Timer.

EJ03: Realizar un variador de velocidad PWM a lazo abierto para un motor DC, que acepte comando del tipo :Dnnn, nnn de 0 a 100%.

Opcional:

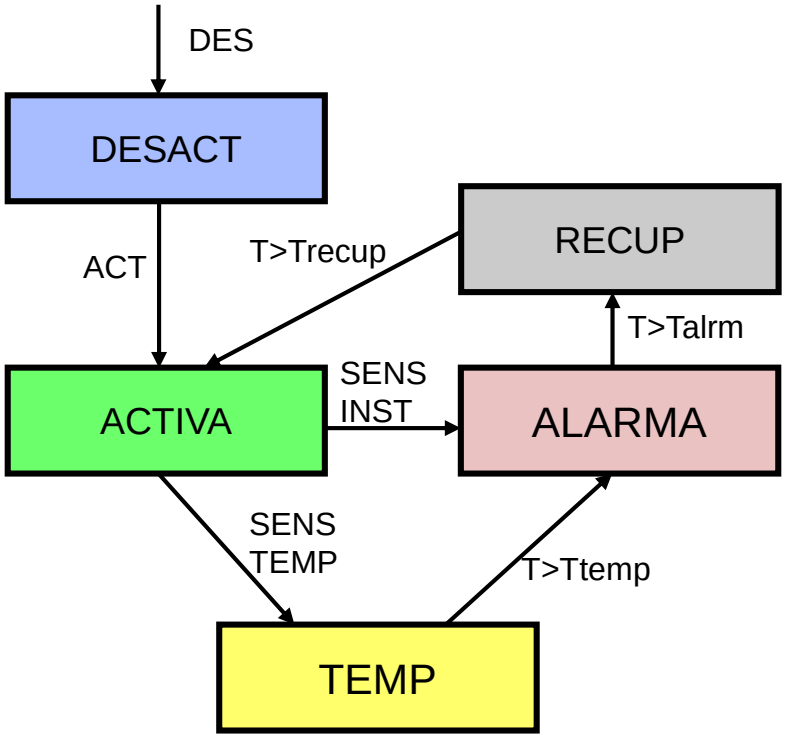
EJ04: Generar en OC1A una onda senoidal de 50Hz con PWM a 10kHz.
Ej05: Generar una segunda onda senoidal en OC1B, en cuadratura con la anterior.
EJ06: Idem ejercicio 3, pero a lazo cerrado (tipo P o PI), midiendo el período de rotación del motor con sensor y módulo Input Capture. En esta caso habrá una consigna de velocidad V.

Máquinas de estado

Máquinas de estado

- Es una técnica eficaz para modelizar y realizar la estructura de sistemas con varios posibles estados o modos de funcionamiento.
- Se realiza el modelo como diagrama de estados.
- Una forma de implementar este diagrama es mediante sentencias “switch/case” que lanzan las operaciones de cada estado, y evaluaciones tipo if o tipo switch/case para producir las transiciones a otro estado (o mediante interrupciones).

Ejemplo: Alarma domiciliaria básica

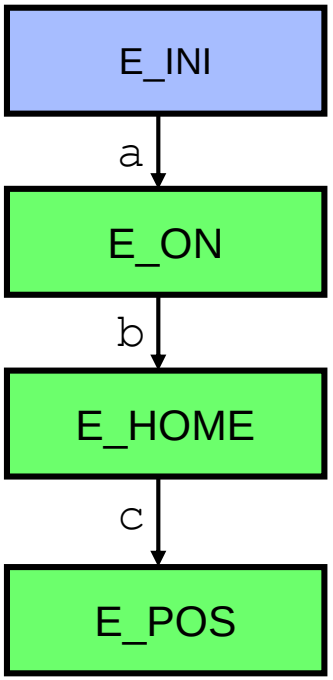



```
enum tEstado{Desact,Activa,Temp,Alarma,Recup};
tEstado estado;
main()
{
estado=Desact;
while(1)
   {
   switch(estado)
      {
      case Desact:
          if (ACT)estado= Activa;
          break;
      case Activa:
          led();
          if (SENS_INST) estado=Alarma;
          if (SENS_TEMP) estado=Temp:
          break;
      case Temp:
          if(T>Ttemp) estado=Alarma;
          break;
      case Alarma:
          alarma();
          if(T>Talrm) estado=Recup;
          break;
      case Recup:
          if(T>Trecup) estado=Activa;
          break;
      default:
          break;
    }}}
```

Nota:

La transición al estado Desact se produce **desde cualquier otro estado** mediante el evento DES, por ejemplo de un pulsador. Es decir, debería agregarse polling de DES en cada estado, o que el evento DES produzca una interrupción, y en dicha interrupción se haga `estado=Desact`

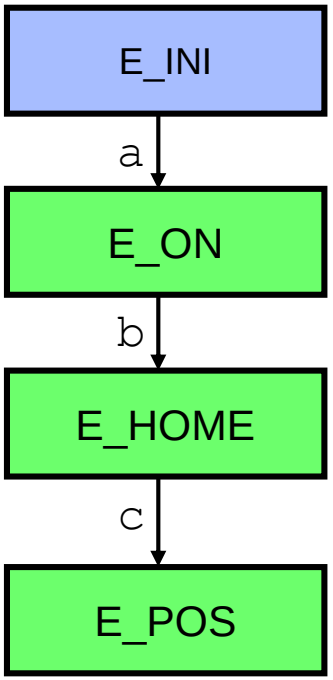
Ejemplo: Estados secuenciales



Es un caso especial en el que los estados se activan en secuencia.

```
enum tEstado{E_ini, E_on, E_home, E_pos};
tEstado estado;
main()
{
estado=E_ini;
while(1)
    {
    switch (estado)
       {
       case E_ini:
          inicializa(); //
          if(a) estado=E_on; break;
       case E_on:
          activa();
          if (b) estado=E_home; break;
       case E_home:
          home();
          if (c) estado=E_pos; break;
       case E_pos:
          pos();
          … break;
       default:  break;
       }
    }
}
```

Ejemplo: Estados secuenciales sincronizados

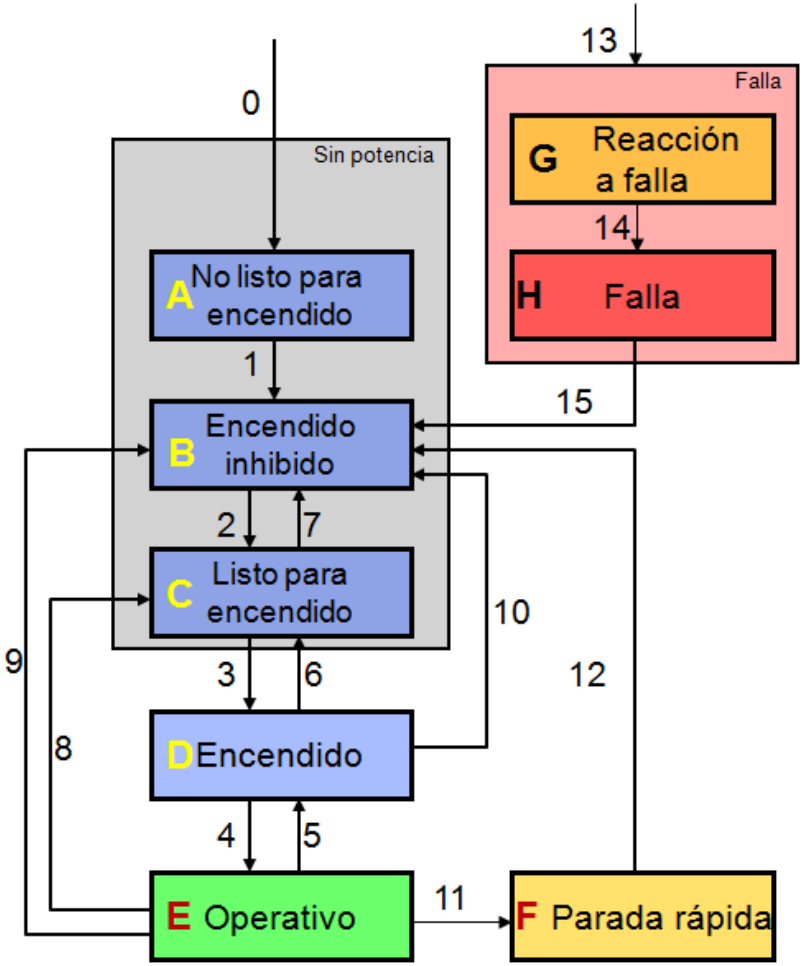



```
ISR(TIMER1_...)
{
sincro=1;
}
```

```
enum tEstado{E_ini, E_on, E_home, E_pos};
tEstado estado;
main()
{
estado=E_ini;
while(1)
   {
   if (sincro)
        {
        switch (estado)
           {
           case E_ini:
              inicializa(); //
              if(a) estado=E_on; break;
           case E_on:
              activa();
              if (b) estado=E_home; break;
           case E_home:
              home();
              if (c) estado=E_pos; break;
           case E_pos:
              pos();
        … break;
           default:  break;
          }
        sincro=0; // es puesto a uno en Int de Timer
   }}}
```

El paso de un estado al siguiente se sincroniza con el período de un timer.
Algunos estados podrían quedar fuera del if, en tal caso su ejecución sería asincrónica.

Ejemplo: Máquina de estados de un servo CANopen

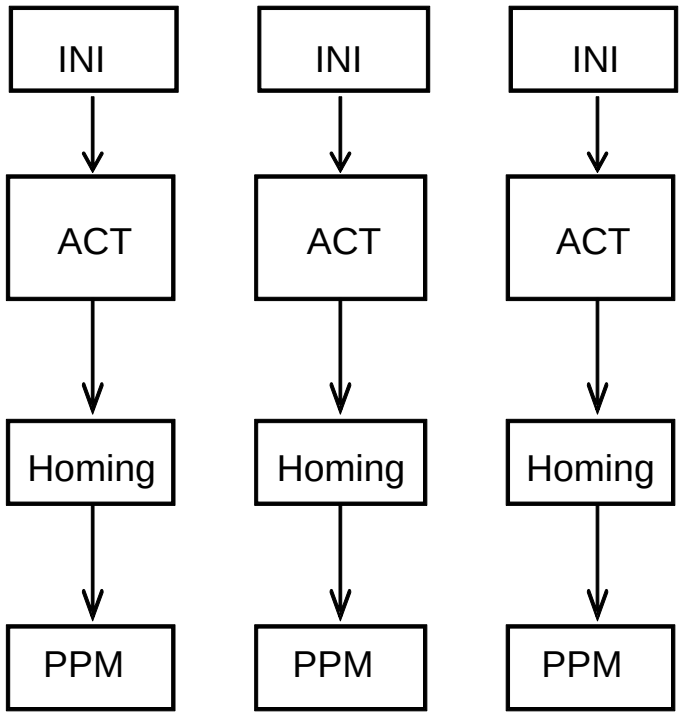


En este caso para la transición entre los estados A, B, C … se requieren condiciones 1, 2, …

Estas condiciones se pueden implementar con **if** o **case**
El estado **A** se activa inicialmente o cuando se dispare la transición 0 desde cualquiera de los otros estados
El estado **G** se activa cuando se dispare la transición 13 desde cualquiera de los otros estados.

Máquinas de estado de ejecución en paralelo

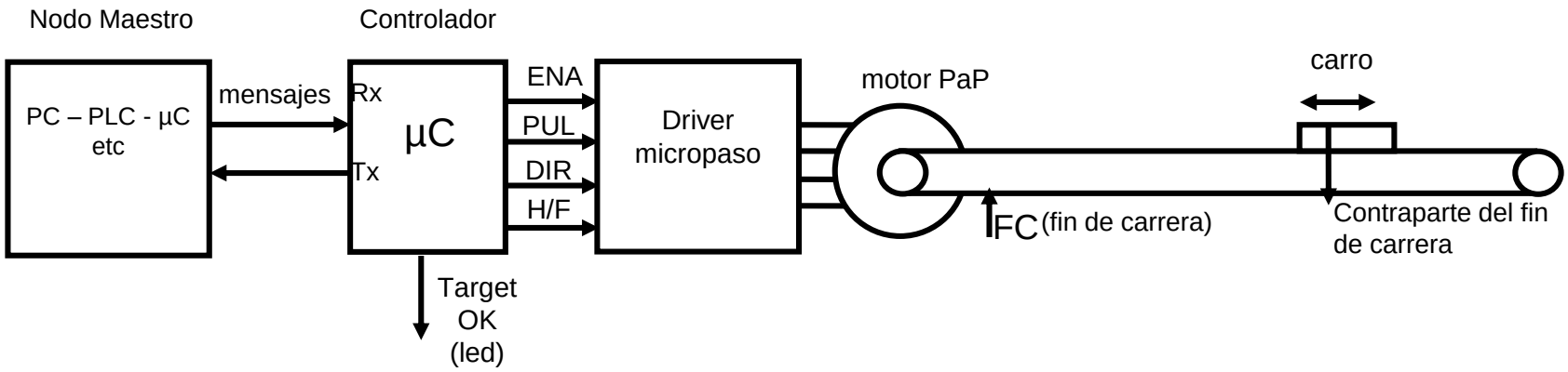
En ocasiones un sistema que actúa como maestro, por ejemplo un coordinador en sistemas multieje, debe supervisar los estados y modos de cada controlador de eje.




```
#define NUMEJES 3
enum tEstado{E_INI, E_ACT, E_HOME, E_POS};
tEstado estado[NUMEJES];

void main()
{
int8 eje;
for(eje=0;eje<NUMEJES;eje++)
   estado[eje]=E_DES;
while(1)
   {
   for(eje=0;eje<NUMEJES;eje++)
      { switch (estado[eje])
         {case E_INI:
            inicializa(eje);
            estado[eje]=E_ACT; break;
         case E_ACT:
            activa(eje);
            estado[eje]=E_HOME; break;
         case E_HOME:
            home(eje);
            estado[eje]=E_POS; break;
         case E_POS:
            pos(eje); break;
         default:  break;
      }}}}
```

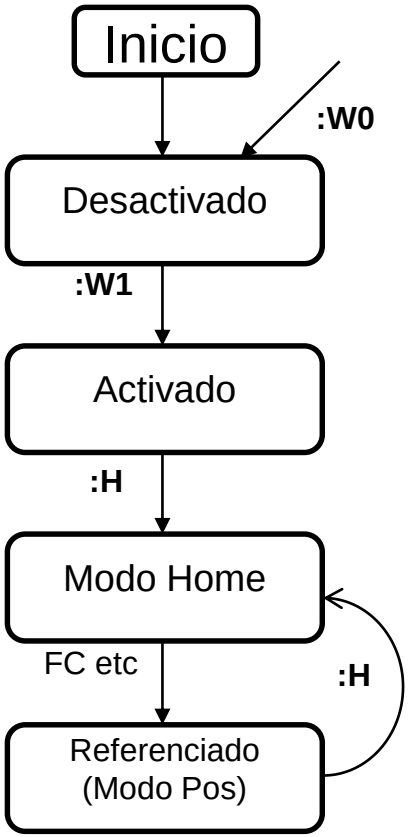
Ejercicio Propuesto



Realizar un control PULSO_DIRECCIÓN de un driver de motor PaP para posicionar un eje lineal que cuenta además con un sensor de fin de carrera para referenciar el origen.

El controlador debe seguir los estados típicos en ejes servocontrolados.
Inicialmente debe estar en **estado Desactivado**, con /ENABLE =1
Un comando **:W1** lo lleva al **estado Activado** , (/ENABLE=0).
Un comando **:W0** lo llevará nuevamente al estado Reposo si fuera necesario.
Una vez activado, el eje debe ser referenciado. Para esto existe una maniobra de "*homing*", la más común realizar un retroceso a velocidad moderada, detener al detectar un cambio de estado en un Final de Carrera **FC** (sensor óptico, inductivo etc), esperar unos 100 ms y avanzar a velocidad muy reducida hasta que el FC **vuelve** a cambiar de estado. Este punto es habitualmente la posición 0, y el eje queda **Referenciado**. Un comando **:H** iniciará la maniobra de homing, que solamente se podrá ejecutar si el eje está activado.

Ejercicio Propuesto (2)



Comandos aceptados

:Pnnn	posición absoluta en pasos.
:Fnnn	avance en pasos
:Bnnn	retroceso en pasos
:Tnnn	tiempo entre pasos, décimas de ms
:Mx	x=0 paso completo, x=1 medio paso (pin 2 de A4988 o DRV8825)
:Ennn	error admisible (en pasos) Por ejemplo :E10 es +/- 10 pasos
:S	Consulta ESTADO. Devuelve palabra de estado en formato Hexadecimal.

Todos los mensajes deben ser interpretados “al vuelo”, debe ser capaz de aceptarlos y ejecutar los durante el movimiento, incluso aquellos que impliquen cambio en la consigna de posición, cambio del tiempo entre pasos etc.

Ejercicio Propuesto (3)

-	-	-	H/-F	PosOK	RefOK	ModHOM	PowON

El controlador dispondrá de un Registro de Estado que podrá ser consultado por el nodo maestro a través del mensaje :S

Bits del Registro de Estado

POWON: En 0 mientras está en estado Desactivado, en 1 en el resto de los estados.
ModHOM: En 1 mientras ejecuta el Homing. Una vez concluido el homing pasa a 0.
RefOK: En 0 mientras no está referenciado, en 1 si está referenciado (al concluir el homing)
H/-F: En 0 para el modo Paso Completo, en 1 para el modo Medio Paso. Inicialmente en 0
okPOS: En 0 mientras la diferencia entre consigna y posición actual sea mayor que el error admisible, en 1 cuando esta diferencia es menor o igual.

Ejemplo: Estados del controlador y comandos para disparar transiciones.

Inicial: Desactivado: fases apagadas, no referenciado. Estado: 0000 0000
Luego de comando **:W1**

Activado: Enable=0, no referenciado, bit PowON en 1 Estado: 0000 0001
Luego de comando **:H**

Modo Home: Secuencia de homing. Activa bit ModHOM. Retrocede con velocidad moderada hasta FC=1, Estado: 0000 0011
detiene, espera 100 ms, avanza con velocidad muy reducida hasta FC=0,
detiene, pone Posición Actual en 0 y bit **RefOK** =1. Cambia a modo Posición Estado: 0000 0101
El bit **PosOK** se pondrá en 0 o 1 según la diferencia entre Pos y PosActual ó 0000 1101

Modo Posicion: Listo para recibir consignas de posición por puerto serie.
En este modo, si abs(posSetP-posActual)<errPos PosOK=1 (*target reached*) Estado: 0001 1101
El estado se transmite en formato hexadecimal. Por ejemplo, el 0001 1011 será "S**1B**"

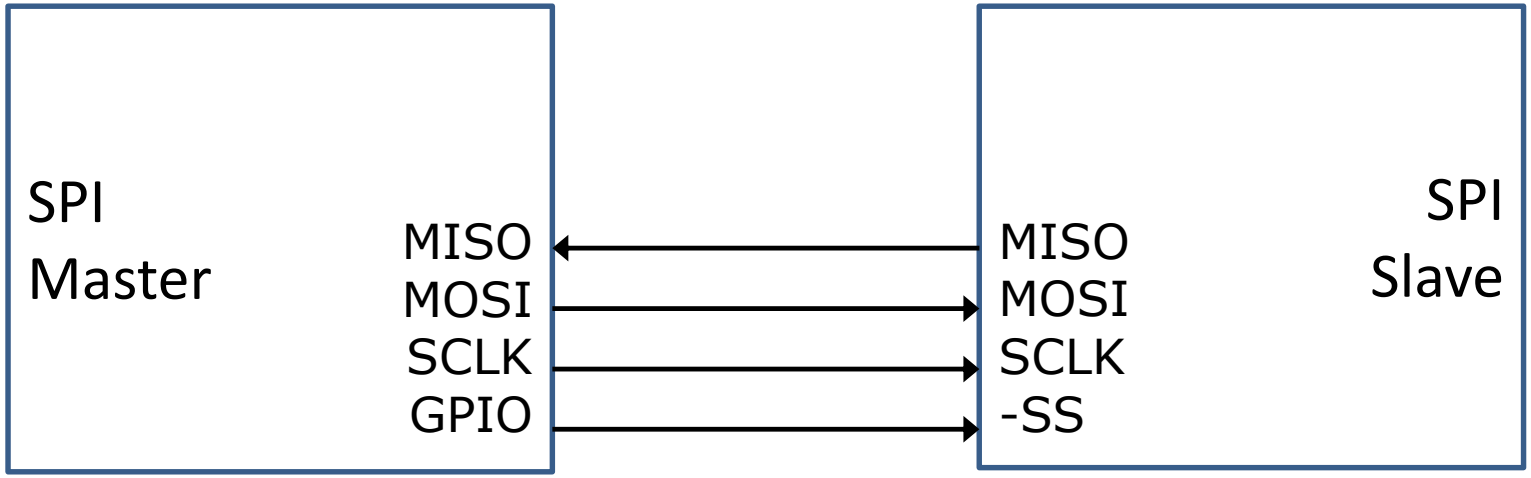
SPI

Interfaces de comunicación serie de microcontroladores:
SPI (*Serial Peripheral Interface*)

- Es un estándar de comunicaciones *de facto*, desarrollado por *Motorola* usado principalmente para la transferencia de información en sistemas embebidos. (*)
- Es una comunicación serie síncrona, Maestro-Esclavo, full duplex, de hardware muy simple. Las puertas de transmisión son *push-pull* (complementarias) lo que permite lograr altas velocidades de transmisión, muy superiores a I2C. No está establecida una velocidad máxima de transferencia, pero en microcontroladores y periféricos estándar es habitual un flujo de datos de hasta 10 Mbps, que en otros sistemas puede ser muy superior.(**)
- Esta interfaz está presente en prácticamente todos los microcontroladores de gama media y alta, y permite comunicarse con una gran variedad de periféricos (conversores A/D, D/A, interfaces Ethernet, CAN, Zigbee, WiFi, memorias Flash, Memorias SD/MMC, sensores MEMS etc).
- En los microcontroladores sin interfaz SPI de *hardware,* puede implementarse una SPI por software (con mayor carga al procesador).

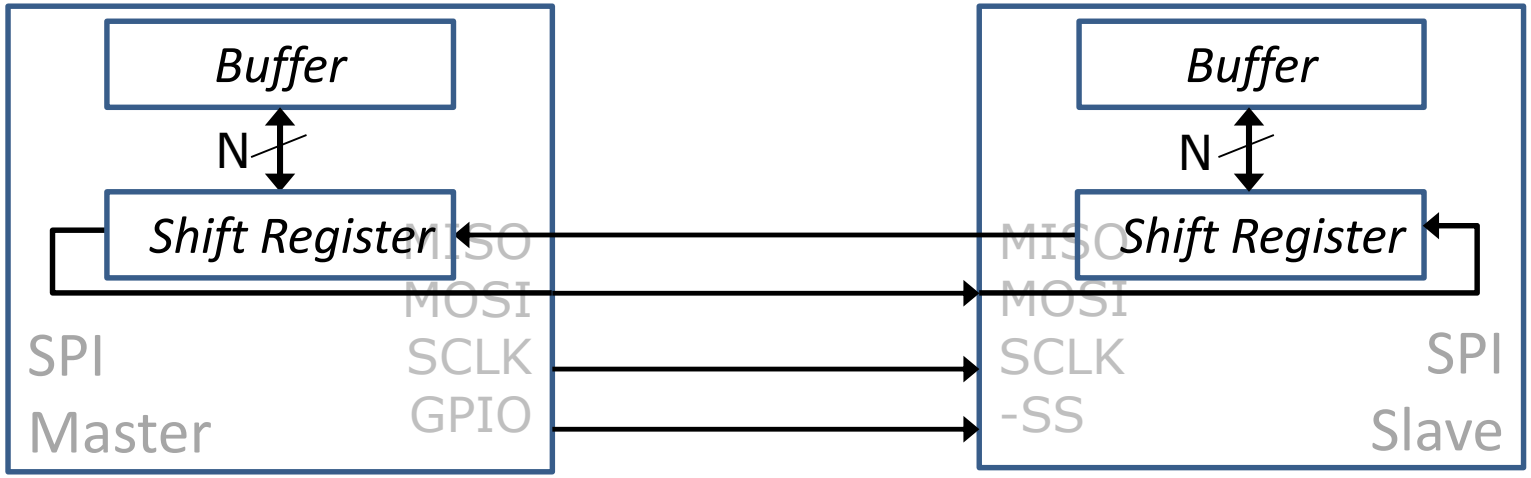
(*) National Semiconductor tiene registradas dos interfaces similares predecesoras del SPI denominadas Microwire y Microwire/Plus.
(**) Hay variantes como Dual SPI y Quad SPI que usan más líneas de datos simultáneas.

SPI: conexión básica entre 2 dispositivos



El maestro maneja el SCLK (serial clock), la salida SDO (ó MOSI Master Out Slave In) y –mediante un pin de salida cualquiera (GPIO, *general purpose I/O*) – la señal de selección -SS (*Slave Select* o *Chip Select*) usualmente activa con '0'.
Cuando –SS está inactiva el esclavo es insensible a las señales en sus entradas SCLK y SDI, y mantiene su salida SDO en ***Tercer Estado***, lo que permitirá una conexión en bus de varios esclavos comandados por el maestro. En una conexión básica con un solo esclavo, -SS se puede conectar directamente a '0' (GND) para que siempre esté activado.

SPI: conexión básica entre 2 dispositivos



Internamente existe un único *Shift Register (*registro de desplazamiento) para cada dispositivo. Estos registros de desplazamiento son habitualmente de 8 bits, aunque hay dispositivos de 16 bits o más. Llamaremos N al número de bits.
Al conectar SDO con SDI y SDI con SDO, quedan en **anillo** ambos registros de desplazamiento. Al cabo de N pulsos de SCLK se produce el intercambio de los contenidos de ambos registros, y en ese instante se vuelcan dichos contenidos a los registros *Buffer*, donde pueden ser leídos por la aplicación.

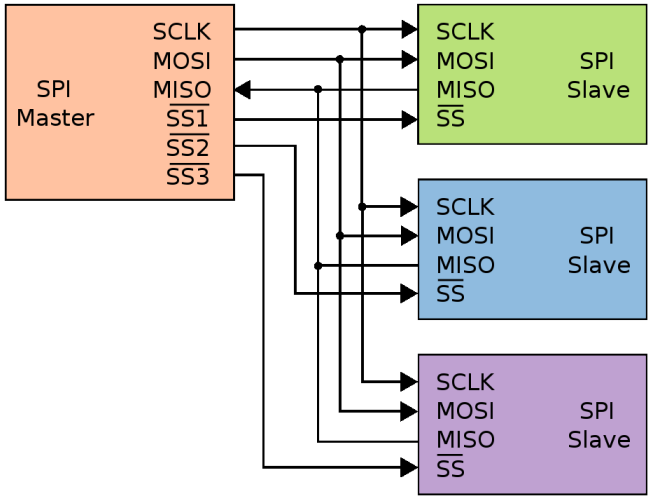
SPI: conexión básica entre 2 dispositivos (2)

Notas:

- Los registros de desplazamiento no son accesibles directamente, sino a través de los *Buffer*, tanto para escritura como para lectura.
- Como se observa, una transmisión implica una recepción y viceversa, por este motivo a la operación se la denomina "intercambio" o "transferencia".
- Si el Maestro quiere **transmitir** dato, lo vuelca a su *buffer*, lo que inicia la transferencia. Puede leer o no el dato que queda en su buffer al final de la transferencia.
- En una operación de **lectura** al Maestro le interesará el contenido que queda en su buffer al final. Para iniciar la transferencia debe escribir un valor *dummy* en su *buffer*. Según el dispositivo esclavo hay valores que pueden estar reservados (pueden ser comandos), por lo que habrá que leer las hojas de datos de los mismos. Habitualmente como *dummy* se escribe un valor 0.

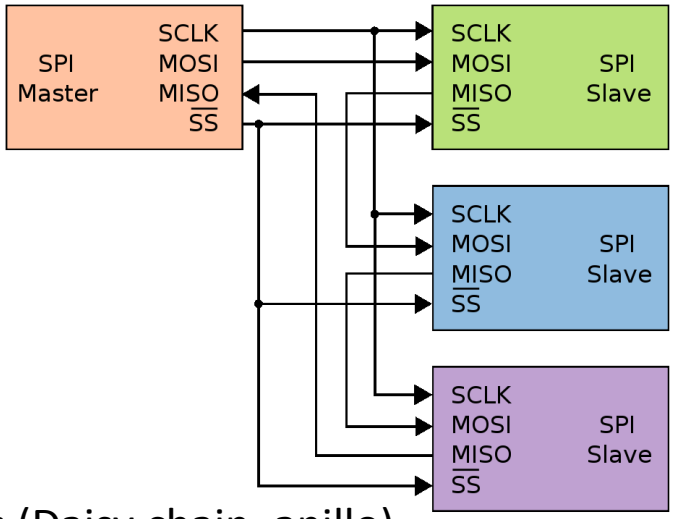
- No se dispone de control de flujo en el protocolo (el Slave no puede avisar si recibió bien, o está listo para seguir recibiendo etc.) aunque esto se puede implementar con pines auxiliares o caracteres de control.
- Cada periférico SPI (A/D, D/A etc.) tiene su propio juego de comandos o caracteres de control.
- En caso de implementar un bus SPI para comunicar varios microcontroladores es posible armar un protocolo propio que incluya sus propios caracteres de control.

SPI: conexión básica entre 3 o más dispositivos



Independientes (Bus)

La SDO del maestro se conecta a las SDI de los esclavos, y las SDO de los esclavos a la SDI del maestro, es decir en bus. Este bus es tomado por el esclavo que es seleccionado mediante su -SS. Para habilitar individualmente (uno por vez) el maestro debe disponer un pin de selección para cada esclavo.
Tiene la ventaja de un acceso inmediato a cualquier esclavo, con la desventaja de usar más pines.
Es útil cuando el acceso a los esclavos va a ser dispar (alguno más atendido que otros) o si los dispositivos no cuentan con ambas puertos (Ej. DACs que solo tienen SDI)



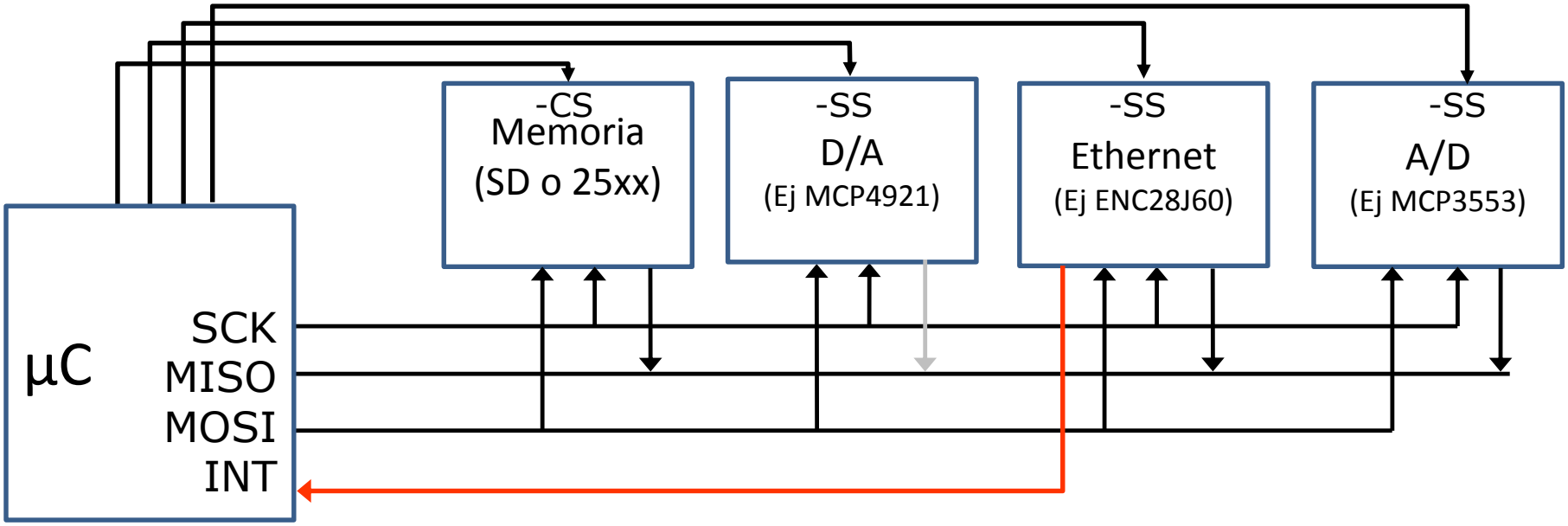
Cooperativos (Daisy chain, anillo)

La SDO del maestro se conecta a la SDI del primer esclavo, la SDO de éste a la SDI del segundo, y así hasta el último.
La SDO del último se conecta a la SDI del maestro cerrando el anillo (si fuera necesario leer los dispositivos).
El bit de selección es común a todos los dispositivos.
Para intercambiar información con los esclavos, el maestro debe poner en '0' –SS, escribir (y/o leer) primero el dato del último esclavo, luego el del penúltimo y finalmente el dato del primero. Este esquema es útil si el acceso es parejo (ejemplo generación de señales simultáneas con DACs, paso de consignas de posición desde el coordinador a controladores de eje de un robot etc.)

En ambos casos la línea SCLK es distribuida del maestro a los esclavos.

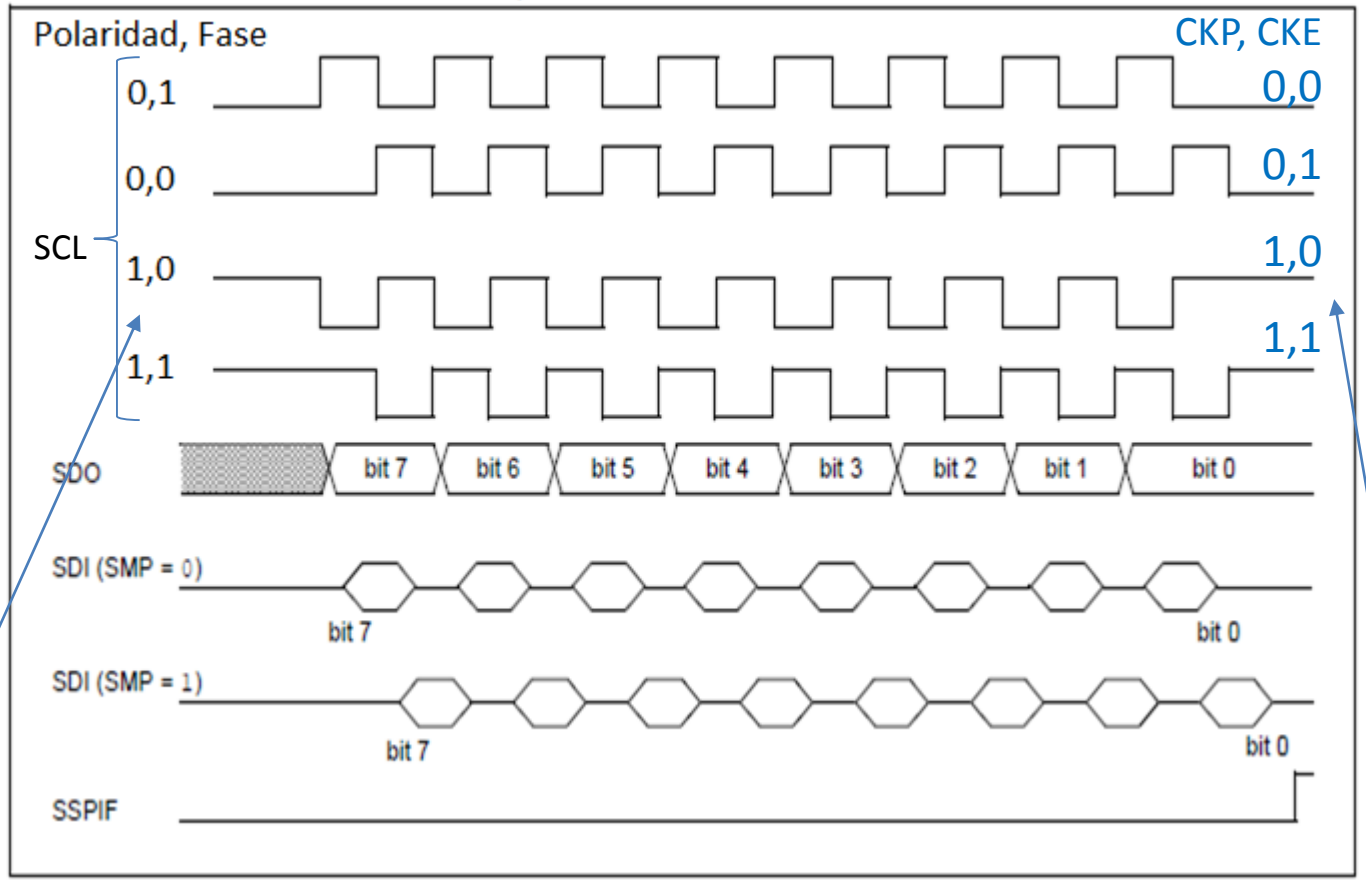
SPI: conexión básica entre 3 o más dispositivos(2)

Ejemplo de un sistema con µC y periféricos



El maestro maneja el SCK (serial clock), la salida MOSI (Master Out Slave In) y las señales de selección SS (*Slave Select* o *Chip Select*, usualmente activas con '0') de cada dispositivo. Algunos esclavos utilizan una línea de interrupción INT para demandar la atención del Maestro, por ejemplo el controlador Ethernet ENC28J60.

Cuatro modos de operación de SPI: Denominación convencional



Denominación convencional

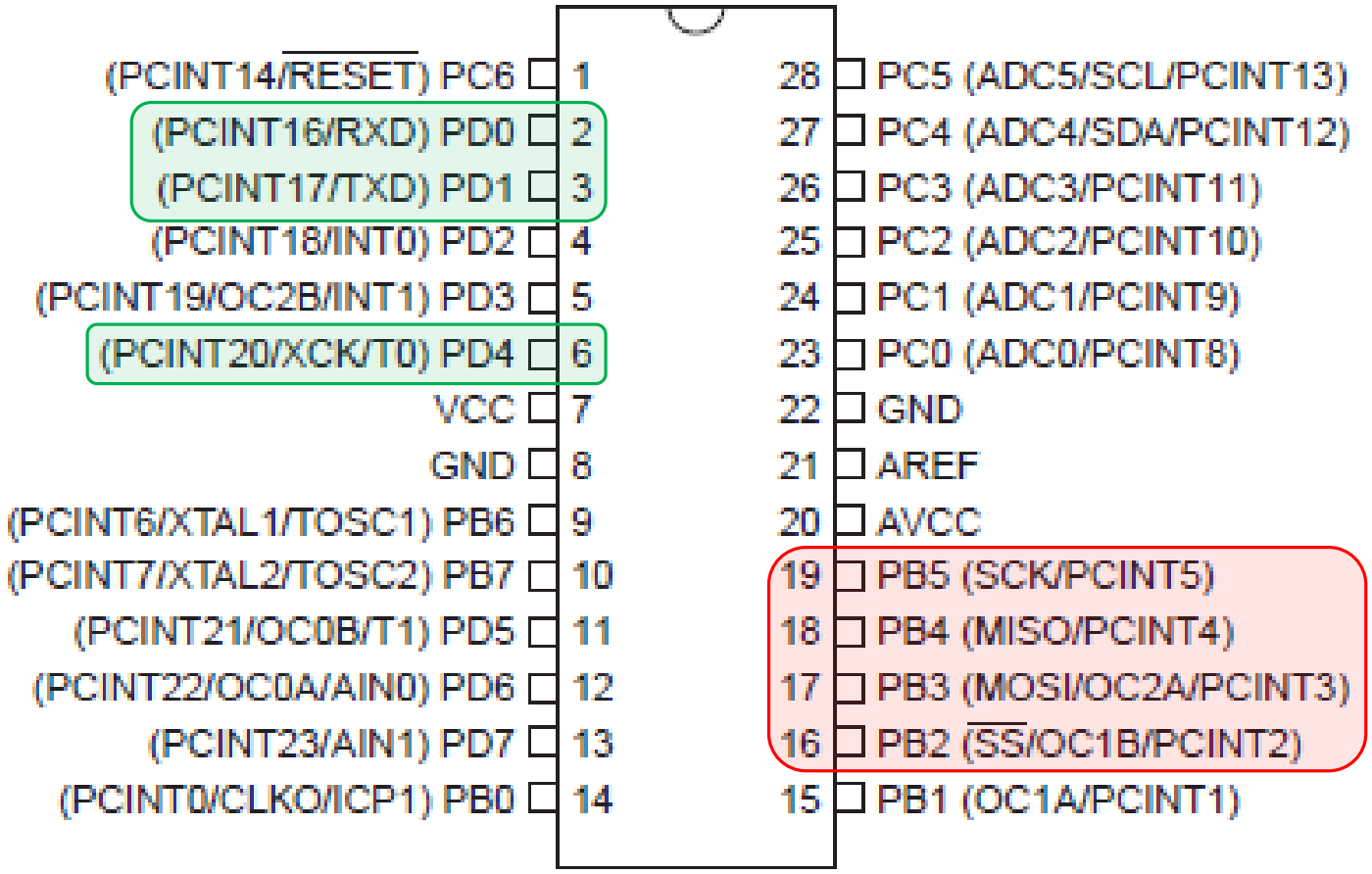
- Polaridad refiere al estado en que queda la línea de CLK entre tramas
- Fase refiere al **nivel** de la señal SCL **al comienzo** del bit de datos.

Denominación en PICs

- CKP es Polaridad y tiene el mismo significado.
- CKE (clock Edge) refiere al **flanco** de la señal SCL que transfiere el dato.

Ej.: Para lograr modo 0,0 debe ser CKP=0 y CKE=1

SPI en ATmega328



El ATmega328p dispone de una interfaz **SPI dedicada** , como Master o como Slave
También puede utilizarse la **USART** como un segundo SPI Master (obviamente toma los recursos de la UART). El pin de clock es en este caso XCK/PD4.

SPI en ATmega

Registro de control SPCR

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
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	

SPE habilita SPI, MSTR es modo master, SPR1:0 prescaler (4,16,64,128)

Registro de estado

Bit	7	6	5	4	3	2	1	0	
0x2D (0x4D)	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SPIF flag de interrupción, wcol flag de colisión, SPI2X=1 duplica velocidad (master)

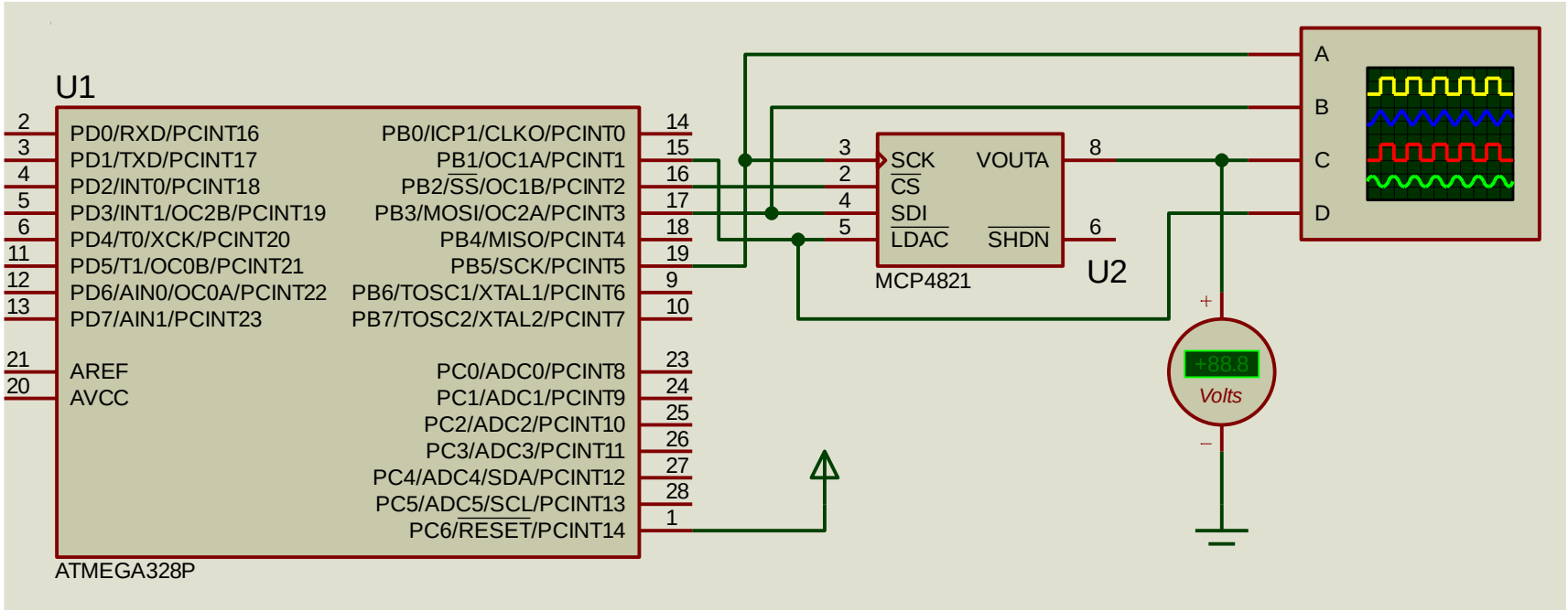
Registro de datos

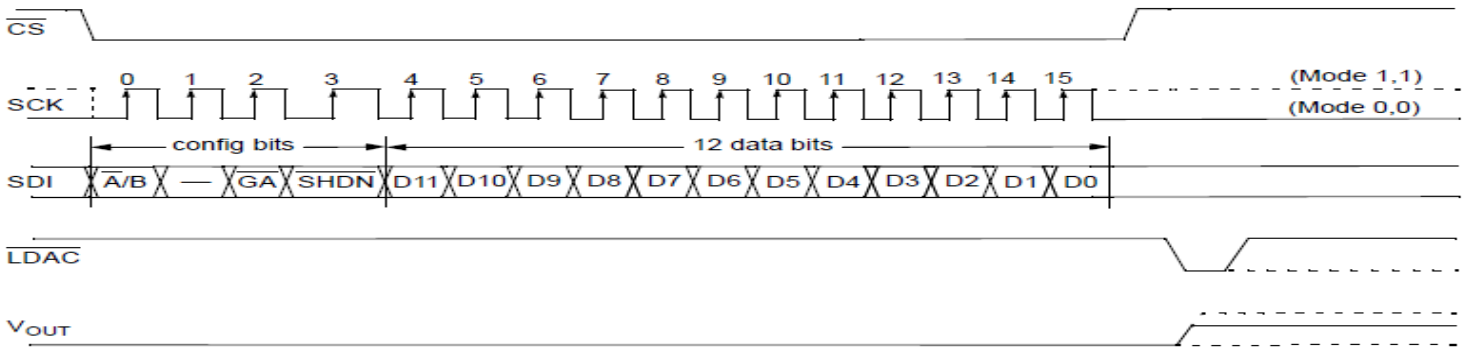
Bit	7	6	5	4	3	2	1	0	
0x2E (0x4E)	MSB							LSB	SPDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	X	X	X	X	X	X	X	X	Undefined

Es el mismo registro para escritura y lectura

SPI: Ejemplos

Ejemplo 01: Escritura de un conversor D/A MCP4821/4921





EJ01_SPI_MCP4821 (ATmega)

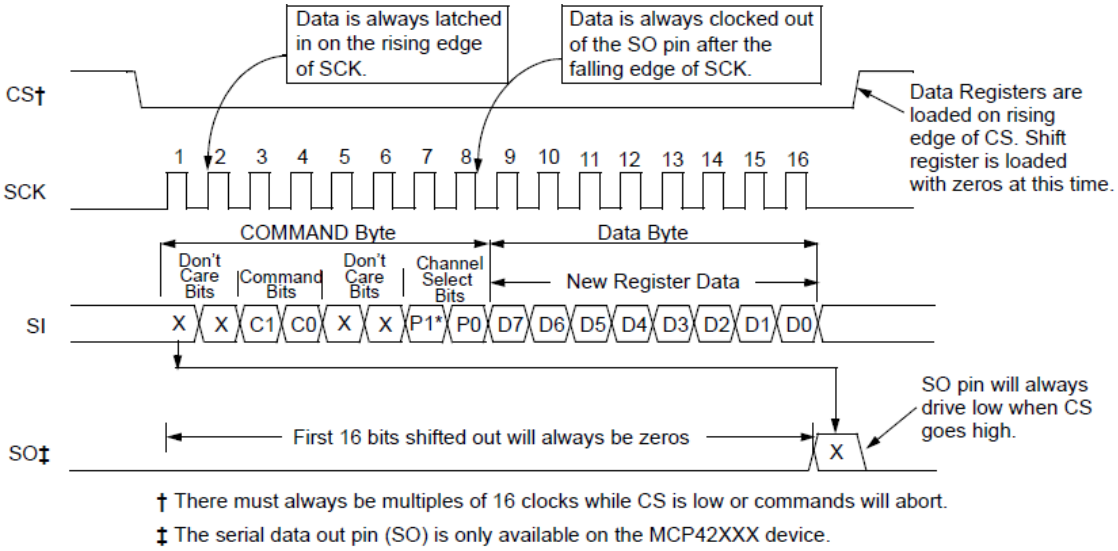
Genera una rampa de 0 a 4095 mV en escalones de 1mV

Ejercicio propuesto 26/5 para **SPI**

Realizar un programa que controle un potenciómetro digital de dos canales, MCP42050.

Opciones:

- Por puerto serie.
- Por pulsadores.
- Secuencia temporizada.



COMMAND BYTE

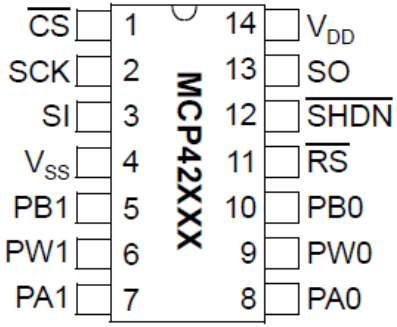
X	X	C1	C0	X	X	P1*	P0

Command Selection Bits

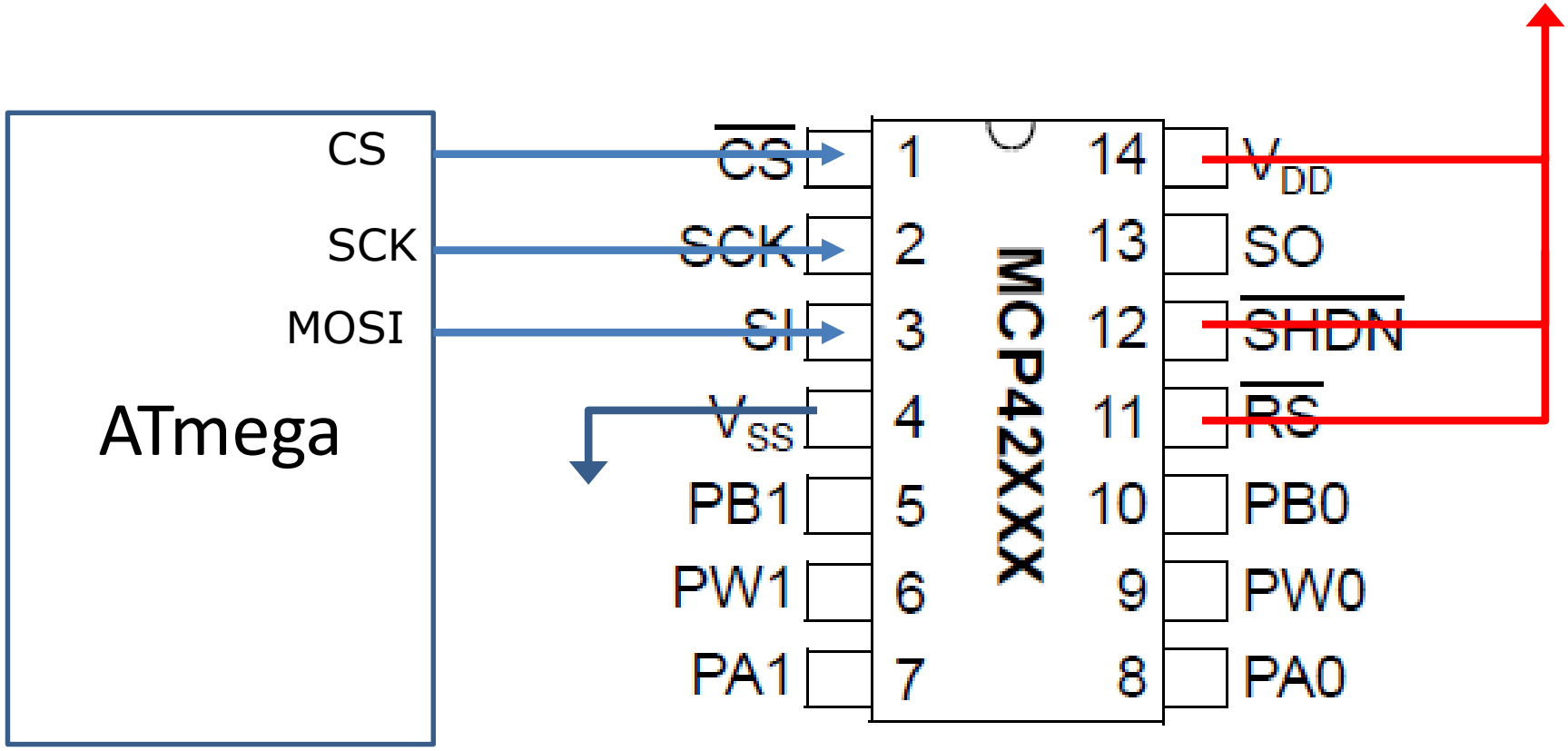
Potentiometer Selection Bits

C1	C0	Command	Command Summary
0	0	None	No Command will be executed.
0	1	Write Data	Write the data contained in Data Byte to the potentiometer(s) determined by the potentiometer selection bits.
1	0	Shutdown	Potentiometer(s) determined by potentiometer selection bits will enter Shutdown Mode. Data bits for this command are 'don't cares'.
1	1	None	No Command will be executed.

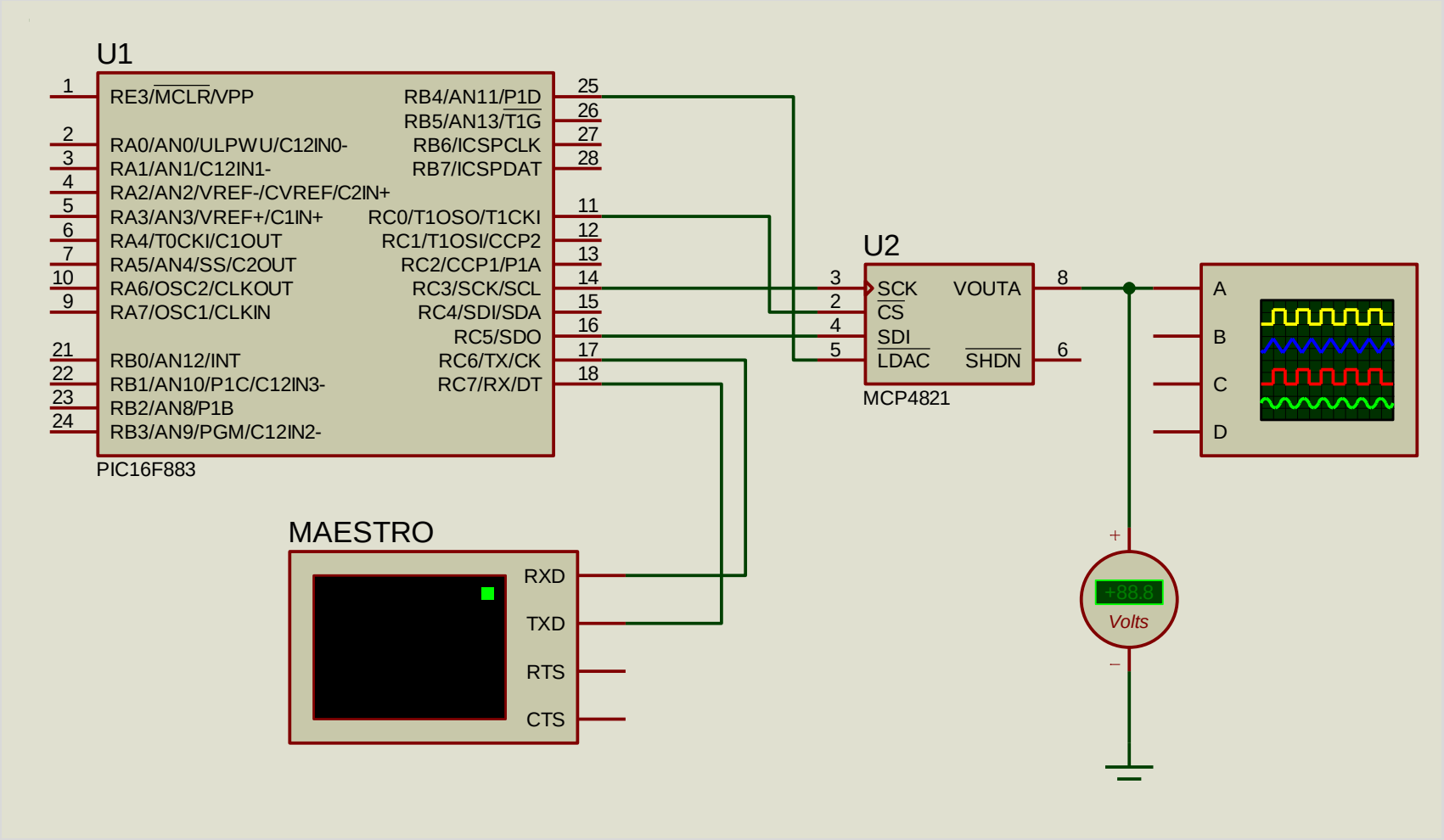
P1*	P0	Potentiometer Selections
0	0	Dummy Code: Neither Potentiometer affected.
0	1	Command executed on Potentiometer 0.
1	0	Command executed on Potentiometer 1.
1	1	Command executed on both Potentiometers.



Ejercicio propuesto 26/5 para **SPI**



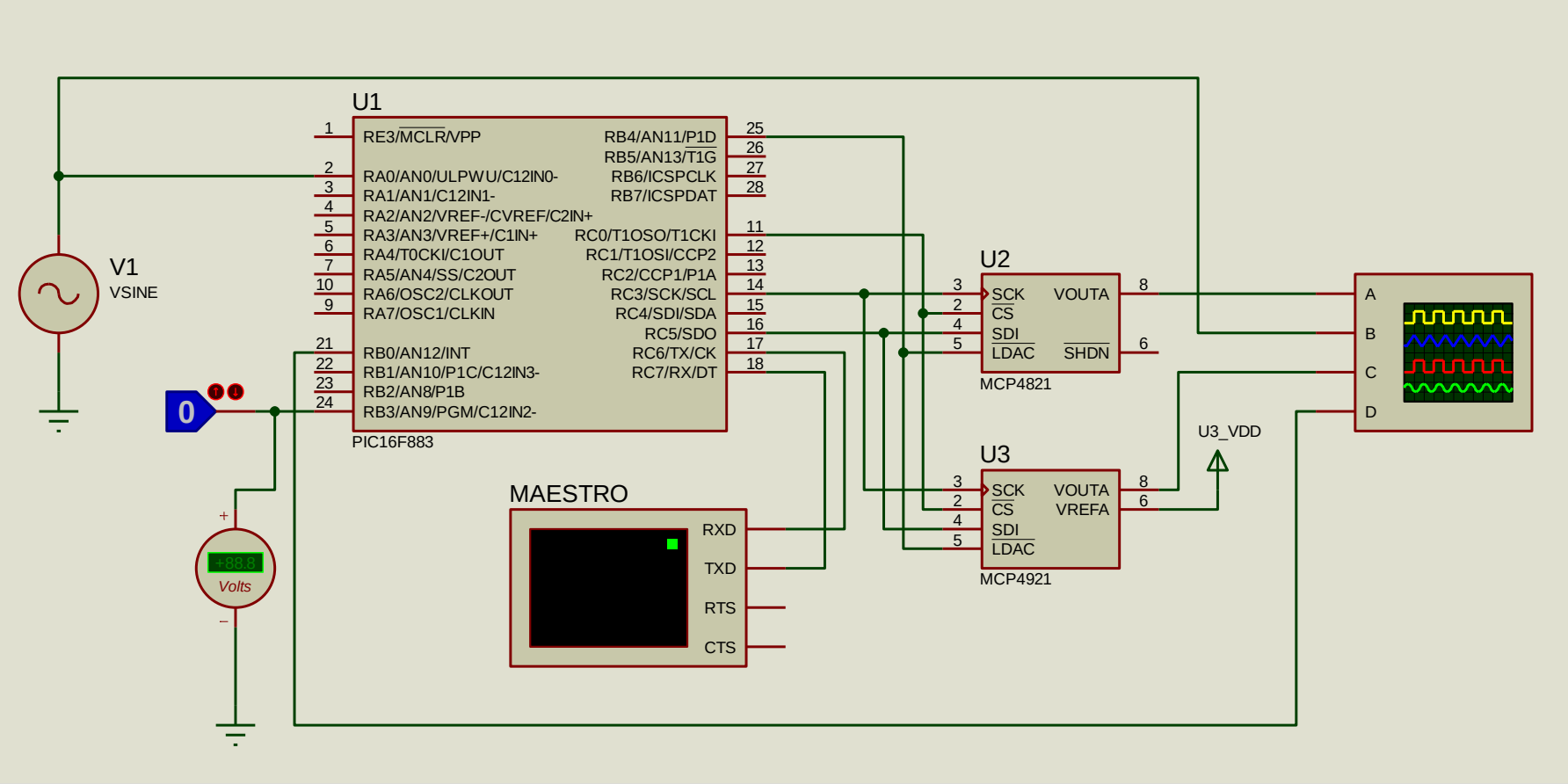
Ejemplo 01 PIC: Escritura de un conversor D/A MCP4821/4921



01_SPI_a_DAC_MCP4821_MCP4921.rar (PIC16F883)

Transfiere valor ingresado por terminal en formato :1,nnnn<enter> al DAC

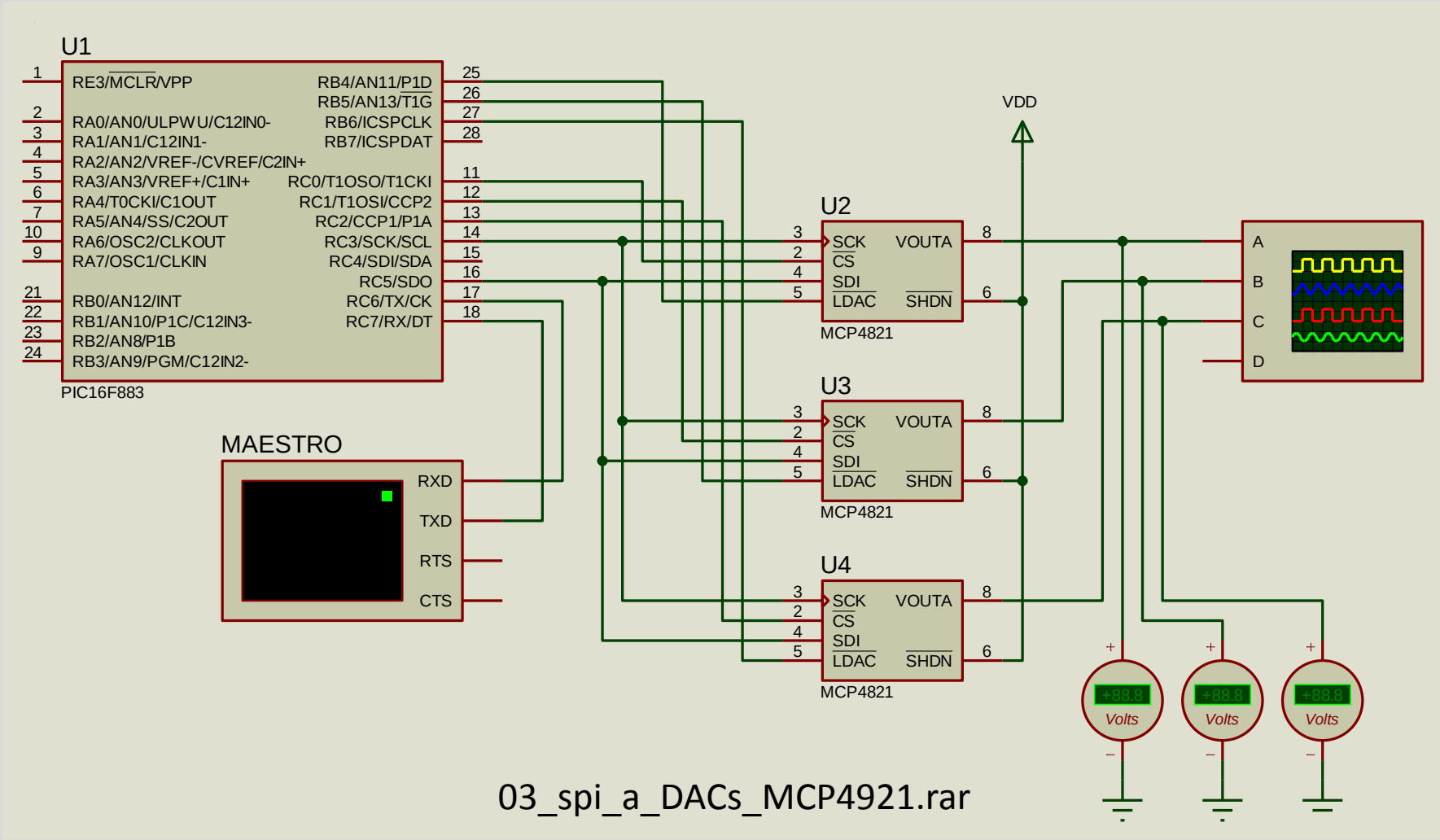
Ejemplo 02: Escritura de un conversor D/A MCP4821/4921



02_SPI_AD_DAC_MCP4821_MCP4921.rar

Idem anterior, transfiere lo ingresado por terminal en formato “:1,nnnn<enter>” al DAC
Si se ingresa “:A<enter>” se pone en “modo analógico”, esto es transfiere la lectura del AD (AN0) a los DACs. Se prueba además cambio de Vref y Ganancia

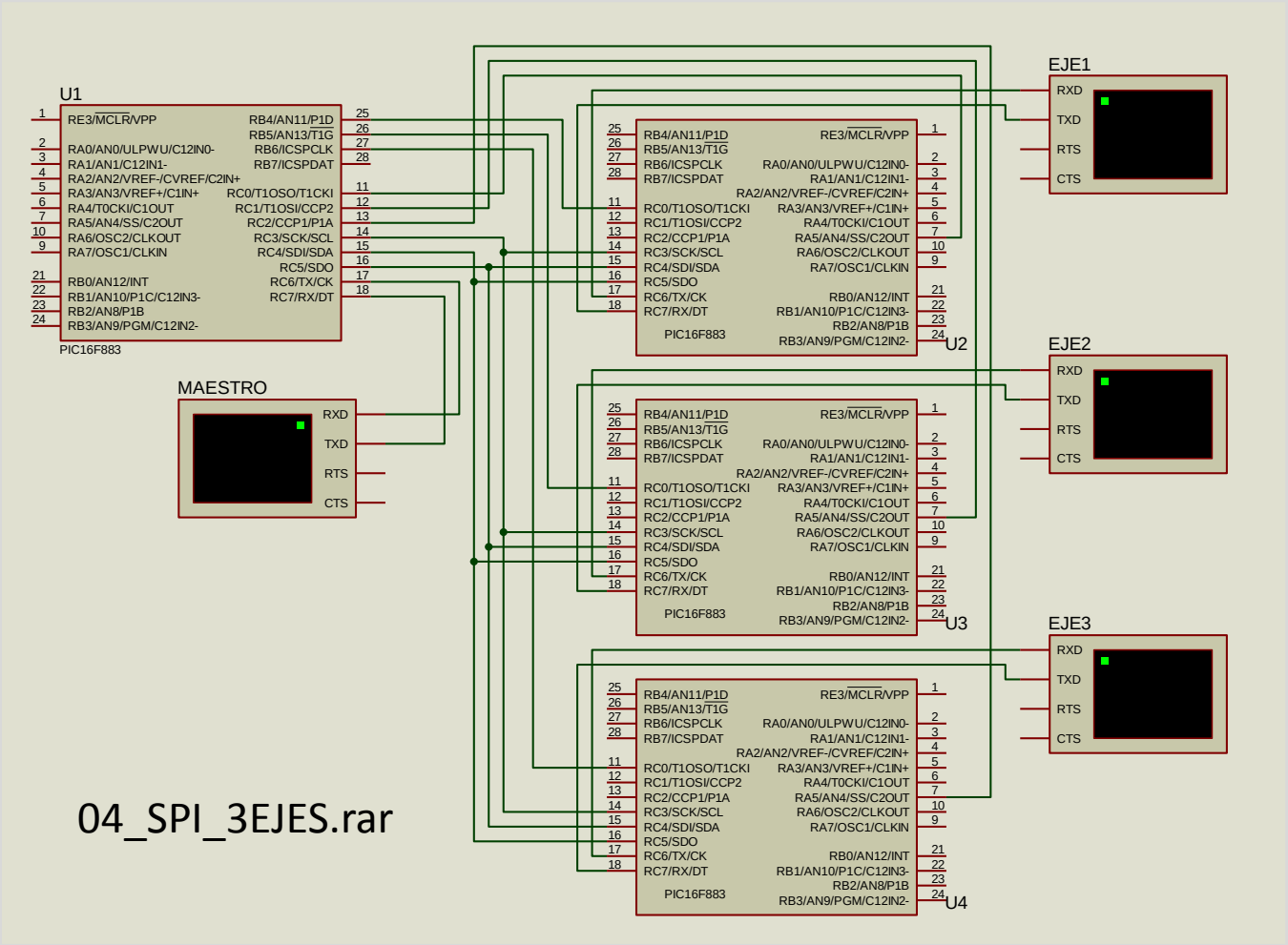
Ejemplo 03: Escritura de conversores D/A MCP4821/4921



Idem ejemplo 1, pero encamina lo ingresado por terminal como “:k,nnnn” con k=1 2 ó 3

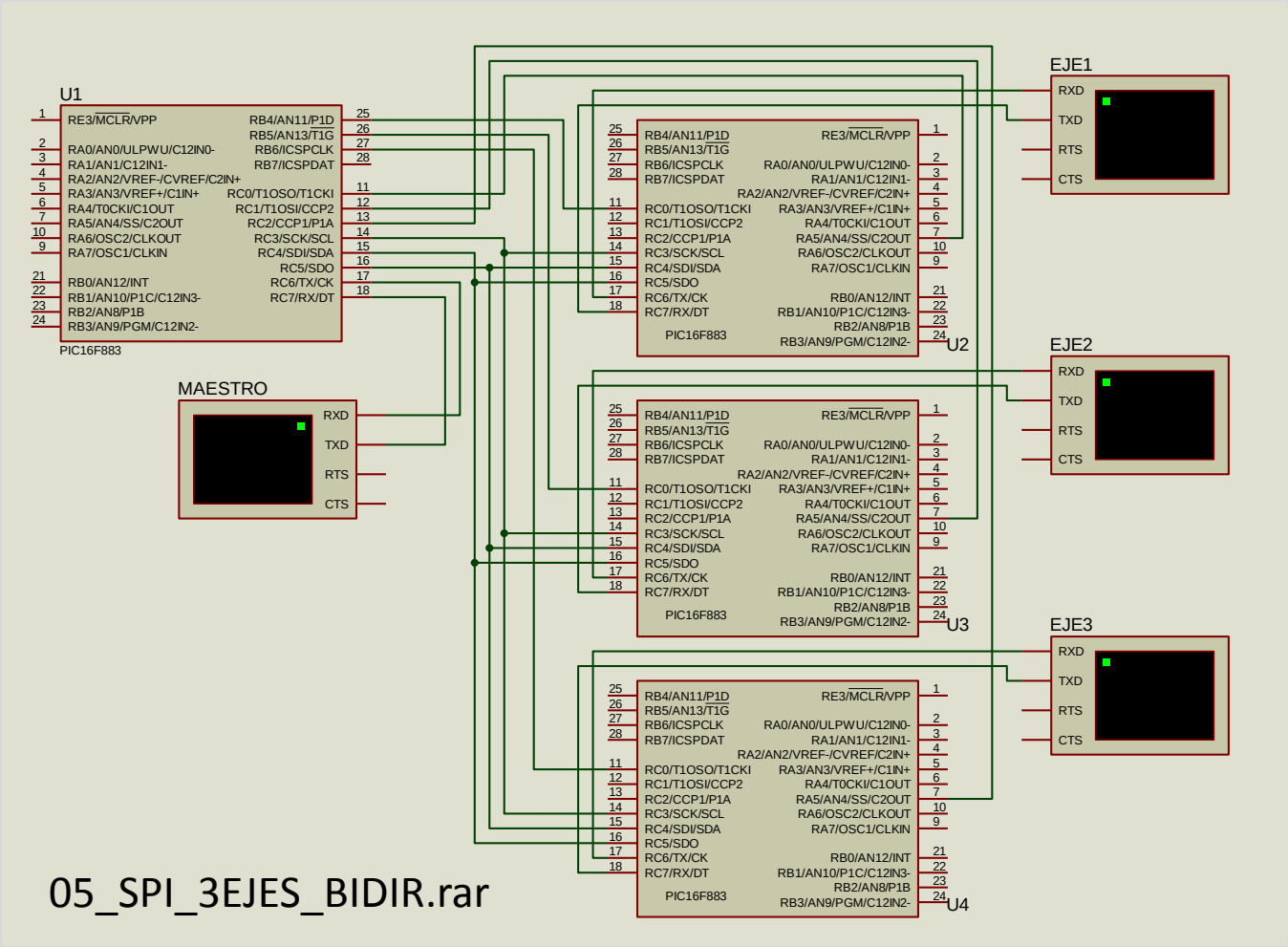
Ejemplo: :3,2814<enter> Pone al DAC 3 en 2814/2=1407 milivolts

Ejemplo 04: Escritura unidireccional a 3 ejes



Idem ejemplo 1, pero encamina lo ingresado por terminal del maestro como “:k,nnnn” con k=1 2 ó 3 a los µC 1, 2 ó 3. Esto se puede ver en los terminales de cada eje.

Ejemplo 05: Escritura bidireccional con 3 ejes



Idem ejemplo 4, pero permite la comunicación desde los ejes al maestro utilizando líneas de interrupción auxiliares sobre RB4, RB5 y RB6 (interrupción combinada “on change”)

Maestro “:eje,dato<enter>” → eje:”dato” Eje “:dato<enter>” → Maestro “eje-dato<enter>”

I2C

I^2C (*Inter IC, IIC…*)

I2C-bus specification and user manual, NXP (antes Philips), UM10204, 2012

Es un bus diseñado por Philips (80's) para comunicación componentes sobre una misma placa.

Originalmemte 100 kbps,
luego "*Fast mode*" de 400 kbps,
desde 1998 "*high speed*" de 3.4 Mbps (con lógica adicional),
actualmente "*Fast Mode Plus o FM+*" de 1 Mbps (sin lógica adicional)

I2C Se puede utilizar para comunicar un microcontrolador con múltiples dispositivos periféricos (memorias, conversores A/D y D/A, relojes calendario, sensores MEMS).

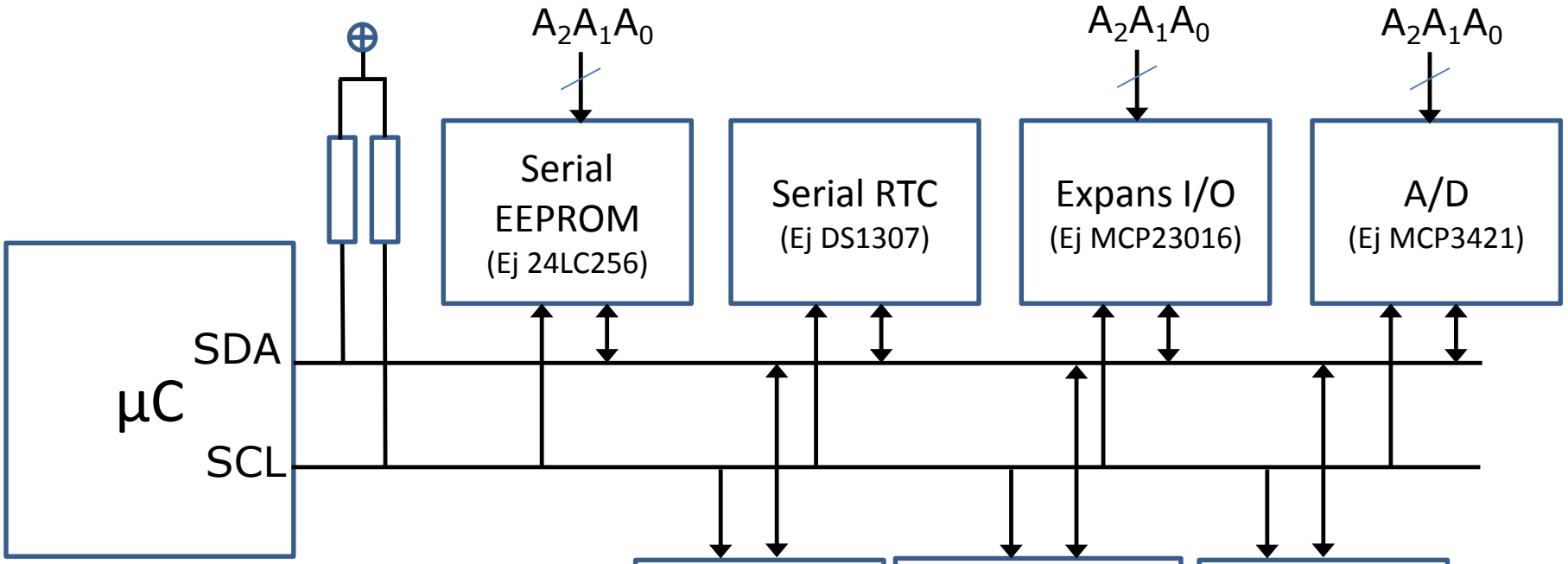
SMBus (*System Management Bus*) deriva de I2C, y es utilizado para monitorear parámetros de equipos electrónicos complejos (Ej. voltajes, temperaturas, velocidad de ventiladores etc en una placa madre). Incorpora restricciones temporales, verificación de error y líneas auxiliares de alerta y suspensión.

PMBus (*Power Management Bus*) es una **capa de aplicación** (7 en modelo OSI) que se apoya en el SMBus (1 y 2 en modelo OSI). Se utiliza para intercambiar parámetros en sistemas de potencia inteligentes (fuentes, inversores etc).

I^2C: Topología y Acceso al medio

- Es una comunicación serie síncrona bidireccional, maestro-esclavo o maestro flotante, con una línea de datos (SDA) y una línea de clock (SCL).
- Eléctricamente es "Colector Abierto", con resistencias de *pull-up en cada línea.*
- En el caso de un sistema maestro-esclavo un único dispositivo (maestro) maneja la línea de clock, iniciando ya sea una escritura o una lectura de parámetros.
- Si es una **escritura**, envía por la línea SDA dirección de esclavo|0, dirección de parámetro y valor de parámetro. En este caso el esclavo solamente produce pulso/s de reconocimiento (ACK o *acknowledgement*).
- Si es una **lectura**, envía por SDA dirección de esclavo|1, y luego continúa con pulsos en SCL mientras el esclavo transmite por la línea SDA el valor del parámetro (que previamente debe haber sido direccionado).

I^2C: Ejemplo de un sistema con µC y periféricos

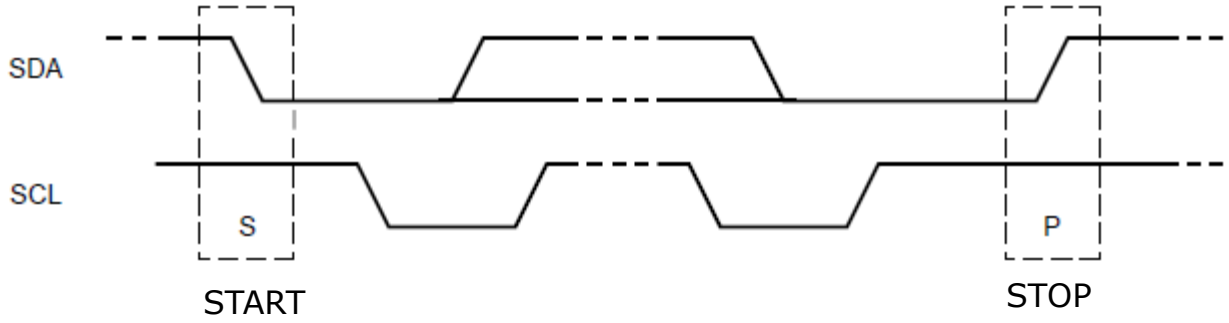


24LC256	1 0 1 0 $A_2A_1A_0$
DS1307	1 1 0 1 0 0 0
MCP23017	0 1 0 0 $A_2A_1A_0$
MCP3421	1 1 0 1 $A_2A_1A_0$
MCP4017	0 1 0 1 1 1 1
MCP4706Ax	1 1 0 0 0 0 x
ADXL345	0 0 1 1 1 0 1 (*)
MPU6050	1 1 0 1 0 0 A_0

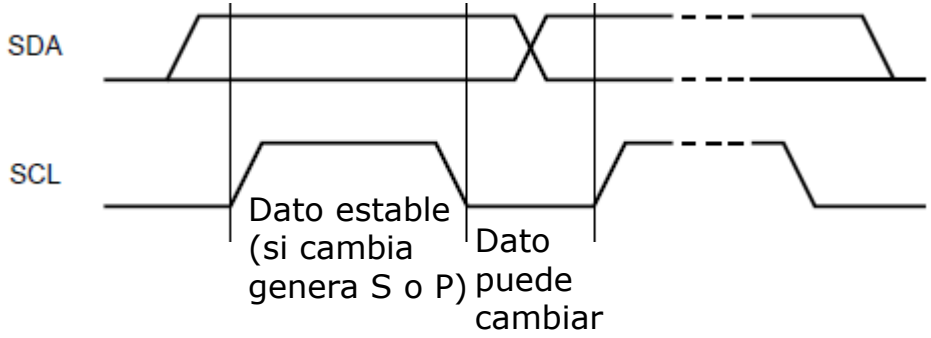
(*) ó 1 0 1 0 0 1 1 según pin ALT ADDRESS

I²C: Tramas

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'. Estos bits son generados por el maestro.



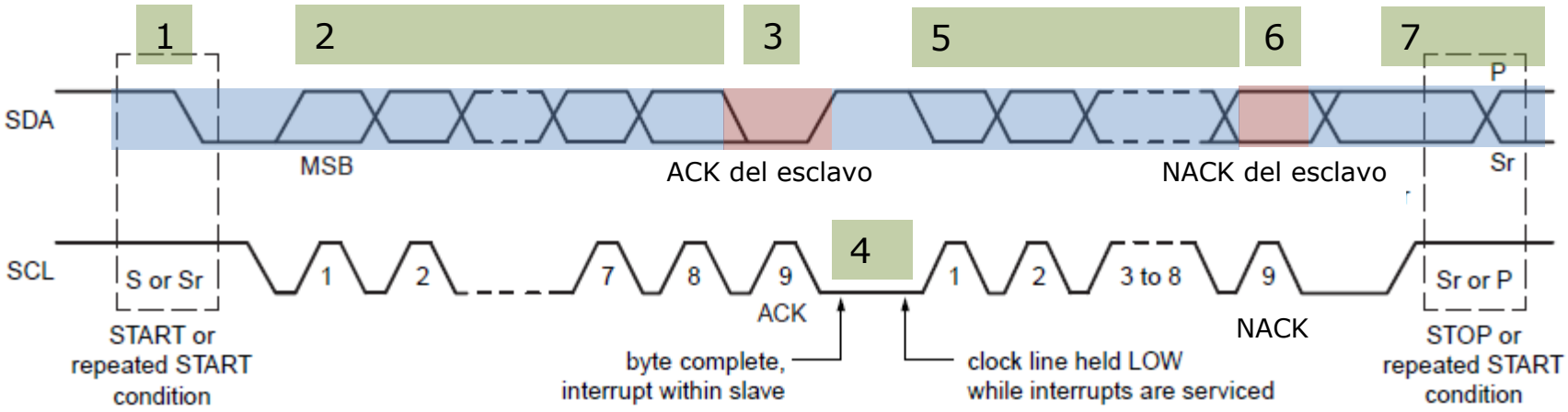
Solamente los bits de START y STOP son cambiantes durante SCL en alto. Los demás bits (de SDA) deben permanecer estables (en '0' o en '1') durante SCL = '1', caso contrario serán interpretados como bits de START o STOP.



Las transacciones son iniciadas por el maestro. El primer BYTE luego de un START consiste en 7 bits para direccionar al esclavo más 1 bit (R/-W). Por ejemplo, para realizar una **lectura** del DS1307 (dirección estándar 1101000), el primer byte es 1101000**1**,y para realizar una **escritura** será 1101000**0**. Los siguientes bytes dependerán de la operación (R/W) que se esté realizando, y son específicos del periférico (A/D, memoria etc). Hasta él próximo START los demás esclavos ignoran los mensajes.

I²C: Etapas de la transacción. Campos de control

1. El maestro Inicia con START,
2. Transmite un byte por SDA (MSB primero) mientras genera los pulsos 1 a 8 en SCL.
3. Genera el pulso 9 en SCL pero libera SDA (colector abierto). Si el esclavo tomó correctamente el byte pone un '0' en SDA durante en este pulso 9 (ACK)
4. Si el esclavo necesita tiempo para procesar el byte recibido, luego del ACK puede poner un '0' en SCL (estiramiento del clock o *clock stretching*). El maestro espera con SCL en colector abierto hasta que la línea SCL vuelve a '1' (el esclavo la libera).
5. El maestro puede transmitir un segundo byte por SDA generando nuevos pulsos 1 a 8 en SCL
6. Nuevamente genera el pulso 9 de SCL pero libera SDA como en 3. Si el esclavo no toma el dato (no está, no pudo, no le corresponde, no lo entiende etc) mantiene el '1' en SDA (colector abierto) durante ese pulso 9.
7. Si el maestro detecta SDA='1' durante el pulso 9 (NACK) puede repetir START o dar STOP.

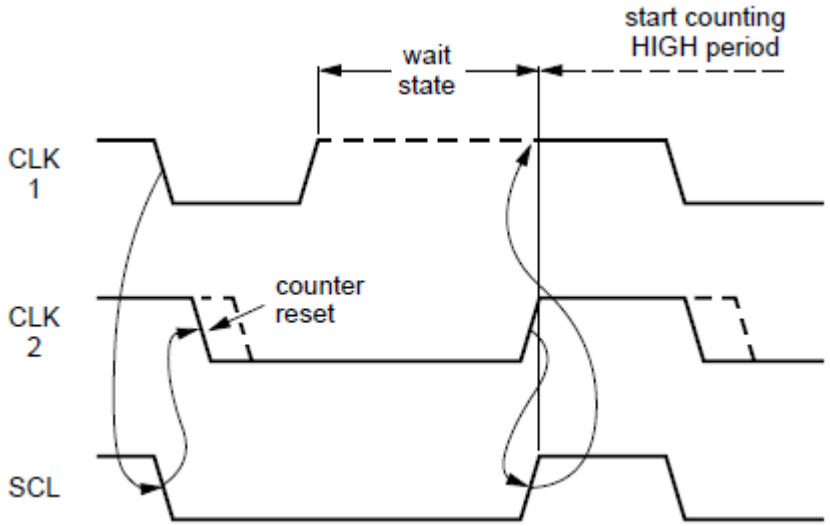


Línea de dato (SDA) tomada por maestro

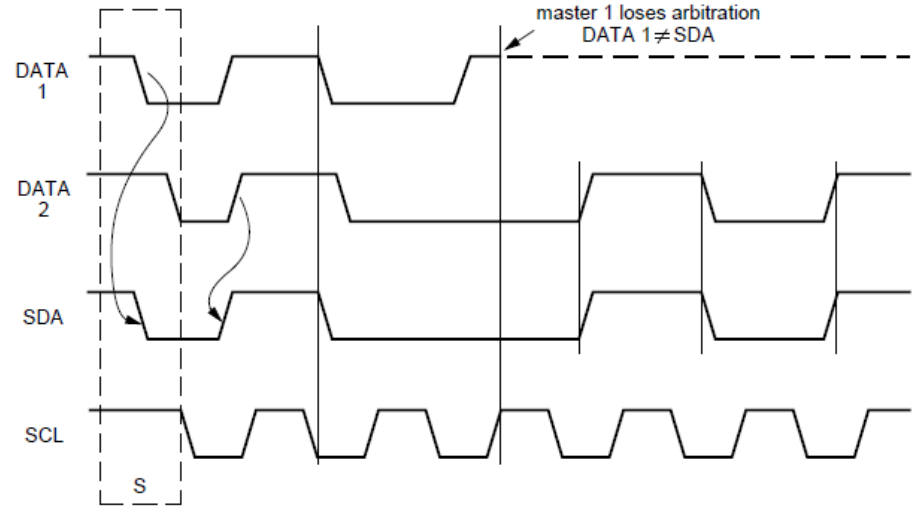
Línea de dato liberada por maestro, esclavo puede tomarla (ACK) o no (NACK)

I²C: Arbitraje multi-master, *clock stretching*, sincronización

Todo se apoya en que tanto en SDA como en SCL el ‘0’ es dominante y el ‘1’ es recesivo (colector abierto = *wired* AND)



Sincronización (multi-master)
Si dos maestros intentan iniciar una transacción al mismo tiempo, toman tanto SDA como SCL. SCL toma como valor la AND entre los CLK generados por cada maestro. Ambos maestros van sincronizando sus datos SDA con el SCL resultante.
La espera automática del maestro es la misma que permite el mecanismo de ***Clock stretching.***



Arbitraje (multi-master)
Si dos maestros iniciaron la transacción al mismo tiempo se van sincronizando. hasta que uno de ellos (el primero que intente poner SDA en‘1’) libera las líneas SDA y SCL.

I²C: Direcciones reservadas

0000 000 0 Mensaje de difusión *(broadcasting o general call address*)
El maestro la envía a todos los esclavos I2C. El byte que sigue puede ser 00000110 (reset y toma de pines de dirección), 00000100 (toma pines de dirección sin reset).

0000 000 1 START byte
Para esclavos lentos (microcontroladores sin hardware I2C dedicado)

0000 001 X CBUS
Para compatibilidad con bus alternativo CBUS, que usa 3ra línea DLEN en vez de ACK.

0000 010 X reservado para otros formatos de bus

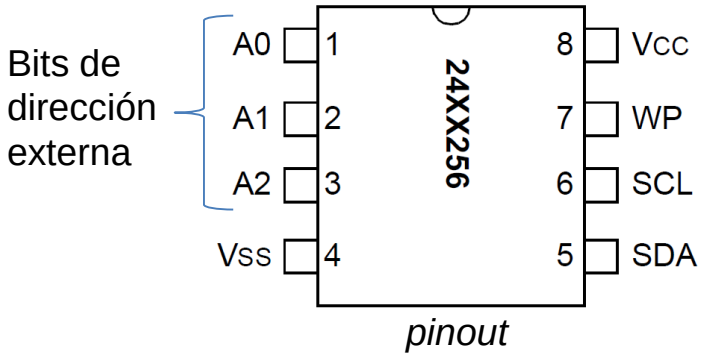
0000 011 X reservado

0000 1XX X código master modo HS

1111 1XX 1 device ID (códigos de identificación de chip para distintos fabricantes y dispositivos)

1111 0DD X **DDDDDDDD** direccionamiento de 10-bit : **DD** son los MSBs de dirección, X es R/-W, el byte siguiente son los **8** bits de dirección restantes (no 7).

Ejemplo: Manejo de memoria 24LC256

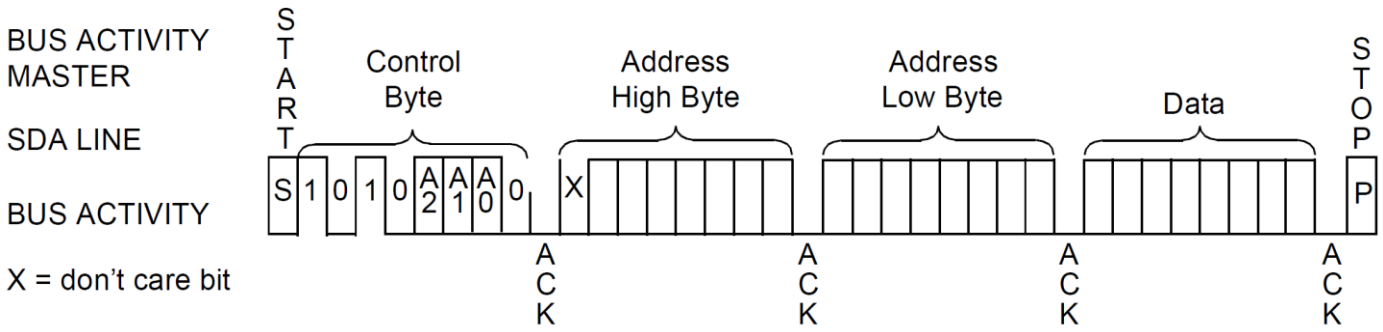


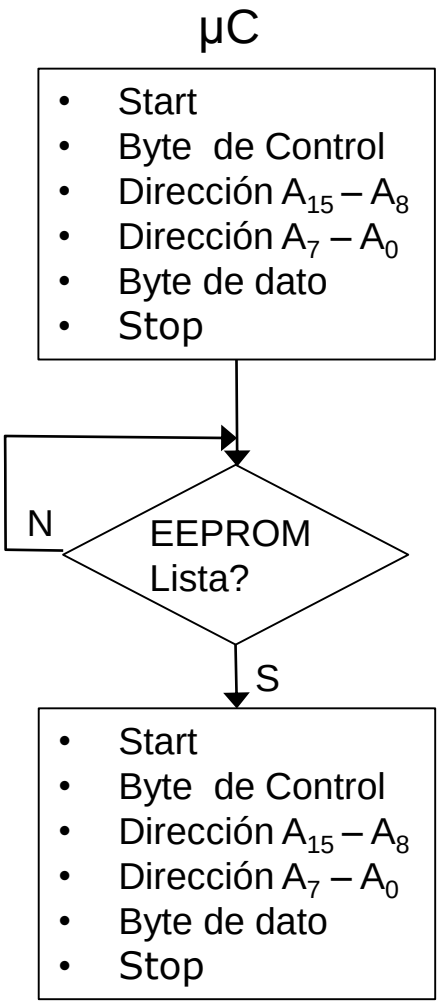
pinout

- 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

Las memorias EEPROM I2C tienen una dirección estándar 1010xxx, es decir si los primeros 4 bits de los 7 bits de dirección son “1010”, se trata de una memoria. Otros periféricos tienen otros primeros 4 bits. Los otros 3 bits 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 1010**000,** y si todos están a Vcc responderá a la dirección 1010**111**

Escritura de un byte en 24LC256





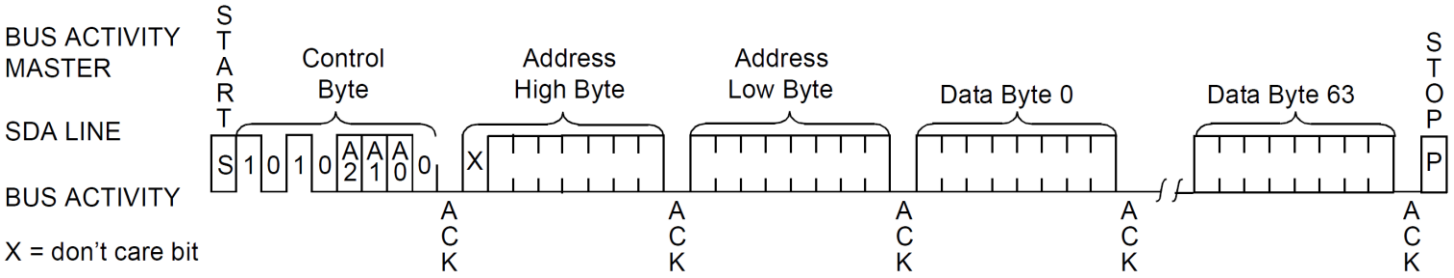
La escritura directa en EEPROM insumiría unos 2 a 5ms por byte. Para acelerar el proceso las EEPROM serie tienen un buffer RAM, en la 24LC256 este buffer es de 64 bytes, que permite una rápida escritura de múltiples bytes, que luego se deberán volcar a la EEPROM "física" en una única operación.
Esta operación de **escritura física** en EEPROM se inicia cuando se envía el STOP, y puede demorar hasta 5ms. Mientras se realiza este proceso la EEPROM no puede recibir ni transmitir datos.
Si se le envía START y el Byte de control mientras no está lista, la EEPROM responde con NACK (nivel alto en SDA).
Para evitar pérdida de datos el µC debe asegurarse de que la EEPROM esté lista.

Se puede hacer de manera simple mediante un *delay* de al menos 5 ms entre un STOP y una nueva operación.

Otra manera, para minimizar la demora, es enviar START y el Byte de Control hasta que se detecta el ACK (EEPROM lista).

Escritura de página en 24LC256

Acceso secuencial de hasta 64 bytes en direcciones consecutivas)



Para acelerar la escritura de bytes en direcciones consecutivas (el uso más habitual) las 24LCxx cuentan con un *buffer* RAM que almacena temporalmente los bytes a escribir.
En la 24LC256 este *buffer* o página RAM es de 64 bytes.
Si se sobrepasa la capacidad de la página se sobreescribe lo anterior (*roll over*).
¡Cuidado!. Los datos son volcados a memoria permanente recién en el STOP del I2C.

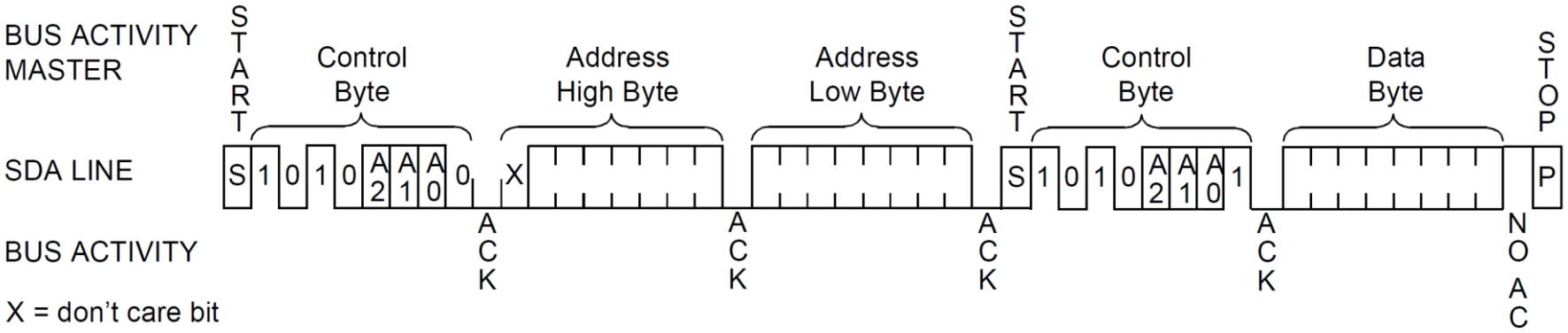
µC

- Start
- Byte de Control
- Dirección $A_{15} – A_8$
- Dirección $A_7 – A_0$
- Byte de dato 1
- Byte de dato 2
- … (hasta 64)
- Stop

Lectura de 24LC256 (1): Acceso aleatorio

La primera parte de la operación de **Lectura** de un Byte de EEPROM se realiza **igual que la escritura,** se escribe el START, el Byte de Control (R/-W=0) y los bytes de Dirección (A15..A0) para direccionar el registro que se quiere acceder. Luego de esto se envía un nuevo START y el Byte de Control (R/-W=1), con lo cual la EEPROM transmite el byte de dato.

µC	EEPROM
• Start	•
• Control (W)	• ack
• Dirección $A_{15} – A_8$	• ack
• Dirección $A_7 – A_0$	• ack
• Start	
• Control (R)	• ack
leer	• Byte (dato)
• Stop	

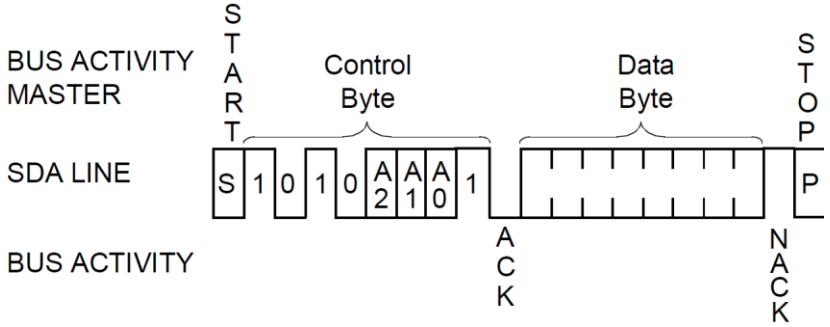


Nota
Un error común es confundir la operación de la Aplicación (lectura de EEPROM, capa 7 del modelo OSI) con la operación del bus I2C (capas 1 y 2 del modelo OSI).
El bit R/-W del byte de Control alude a la operación I2C, no a la Aplicación.
Por eso la trama vista presenta START, Control (W), Escritura de A15-A0, START, Control(R), Lectura de Byte

Lectura de 24LC256 (2): Dirección actual y acceso secuencial

Lectura de un byte (dirección actual)

La EEPROM retiene en un contador interno la útima dirección accedida (en escritura o lectura) incrementada en 1.
Es posible entonces un acceso inmediato al dato apuntado

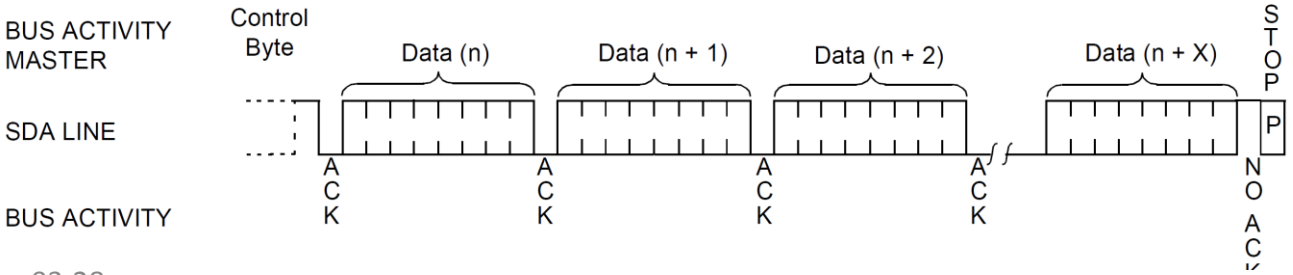


µC	EEPROM
• Start • Control (R) • *leer* • Stop	• • • Byte (dato) •

Lectura N bytes (secuencial)

Como el contador interno se incrementa luego de cada lectura (o escritura) pueden leerse direcciones consecutivas, desde 0 hasta la última (no existe el límite de la página RAM de 64 bytes como en la escritura)

µC	EEPROM
• Start • Control (R) • *leer* • *leer* • … • Stop	• • ack • Byte (dato) • Byte (dato) • … •



Interfaz I2C en ATmega

En Atmega la interfaz I2C se denomina TWI (*2-wire* interface), por cuestiones de patentes, pero es la misma, con las siguientes funcionalidades básicas:

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

Interfaz I2C en Atmega - registros

Registro de Bit Rate (para modo Maestro)

$$SCL\ frequency = \frac{CPU\ Clock\ frequency}{16 + 2(TWBR) \cdot (PrescalerValue)}$$

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

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

Start: TWINT|TWSTA|TWEN
Trans Address: TWINT|TWEN
Escribe Dato: TWIN|TWEN
Lee Dato: TWIN|TWEN|[TWEA]
Stop: TWINT|TWSTO|TWEN
Hab interr: TWIE
Acknowledge al leer, en TWEA

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

Prescaler x1, x4, x16, x64
Códigos de estado (msk 0xF8)

START	0x08
REP_START	0x10
SLA_ACK	0x18
SLA_NACK	0x20
DATA ACK	0x28
DATA NACK	0x30
STOP	0xA0

Registro de Datos

Bit	7	6	5	4	3	2	1	0	
(0xBB)	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0	TWDR
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	1	

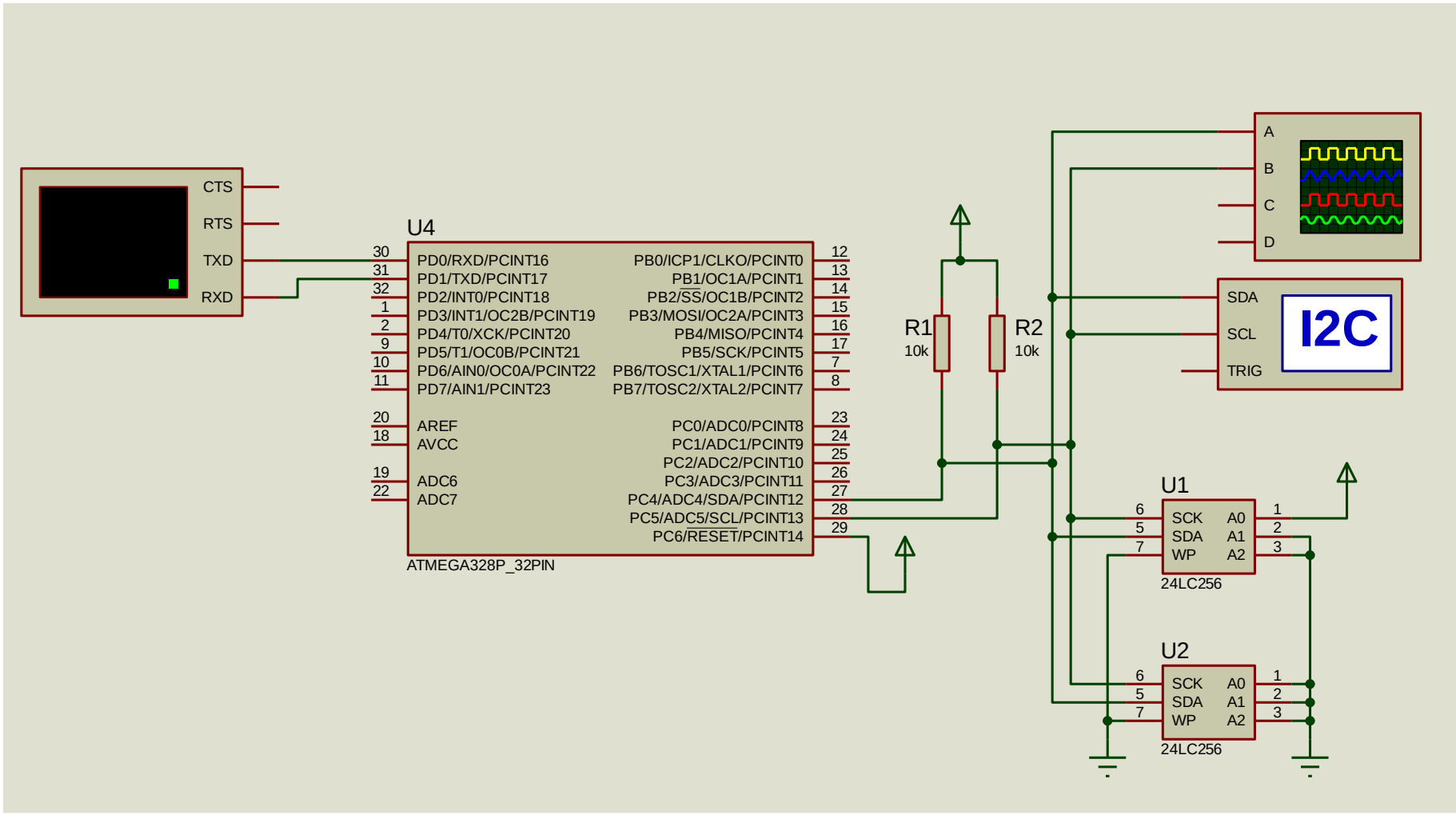
Registro de Dirección y registro de máscara de dirección (para modo esclavo)

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	

I2C: Ejemplos

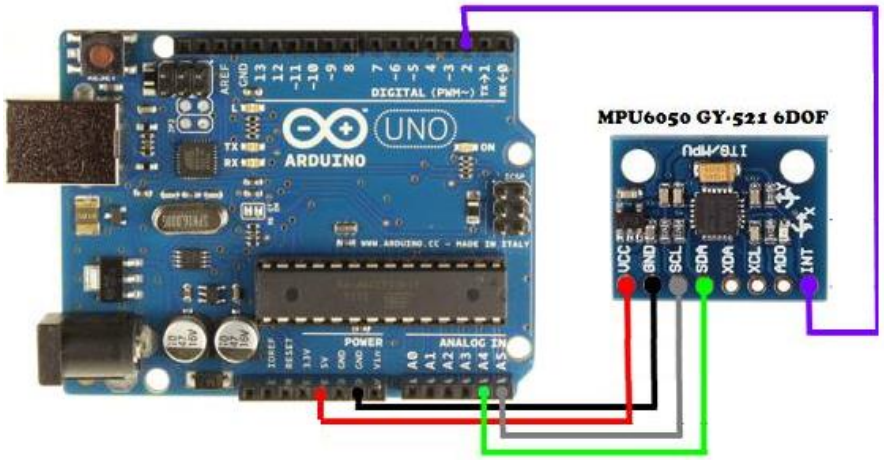
Ejemplo 01 – Acceso a una EEPROM 24LC256



Ejemplo 02 – acceso a MPU6050

i2c_master.c

i2c_init(*uint32_t* F_SCL) Inicializa i2C de uC. (bps de SCL)
i2c_start(uint8_t address) START y dirección de esclavo
i2c_write(uint8_t data) escribe en esclavo
i2c_read(uint8_t ack) lee de esclavo (y pone ack o nack)
i2c_stop(void) STOP



i2c_dev.c

dev_ready(*uint8_t* ad0)
dev8_write8(uint8_t ad0, uint8_t address, uint8_t dato)
dev8_read16(uint8_t ad0, uint8_t address) // lee 2 bytes consec
dev_read_fast(uint8_t ad0) // lee en pos actual

Mpu_6050.H

Definición de direcciones de registros internos del MPU, por ejemplo
#define MPU6050_RA_ACCEL_XOUT_H 0x3B
…
#define MPU6050_RA_GYRO_ZOUT_L 0x48

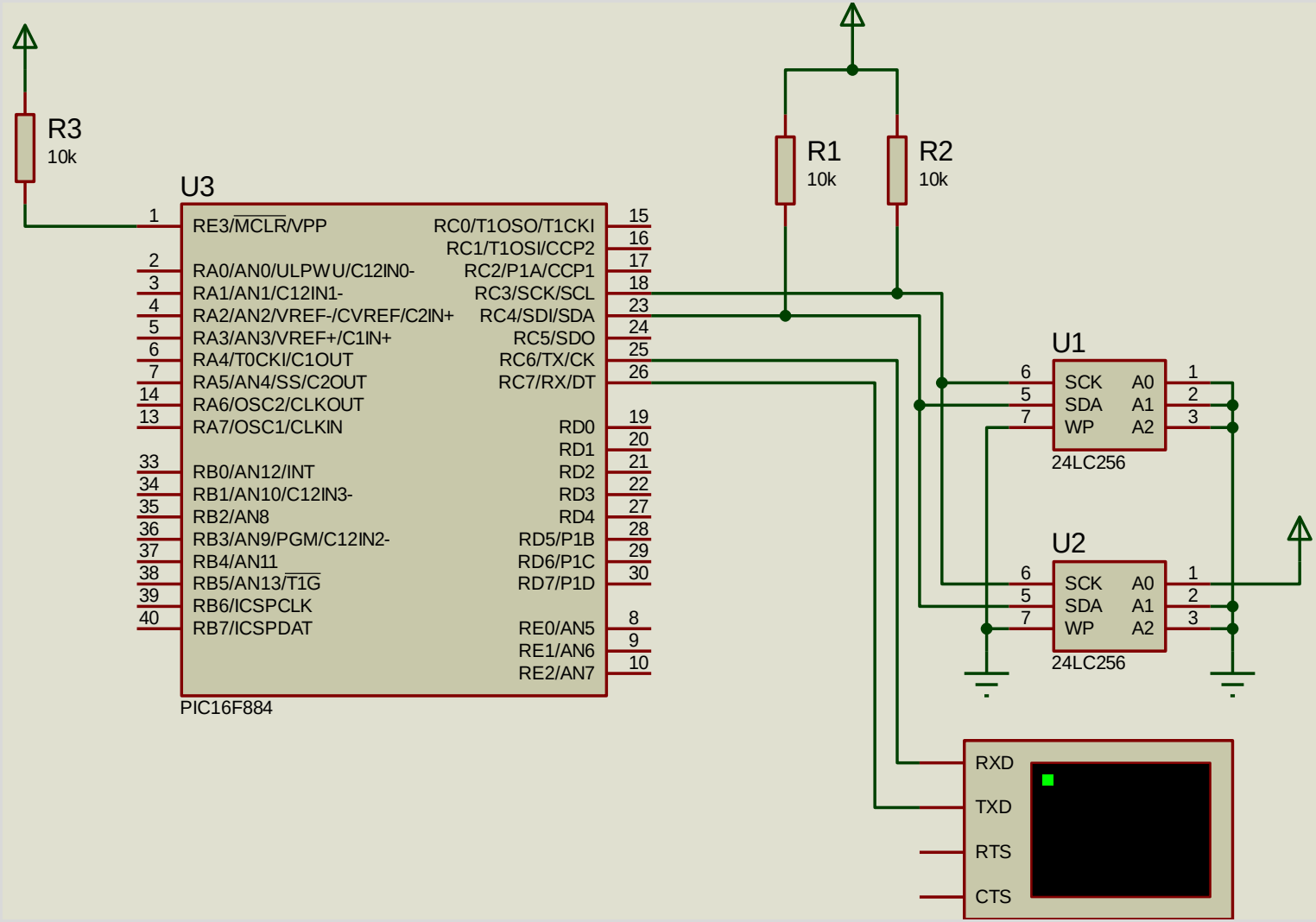
Aplicación: Manejo de un acelerómetro-giróscopo MPU

```
i2c_init(100000UL);                                  // enciende el i2c del uC
dev8_write8(0,MPU6050_RA_PWR_MGMT_1, 0x0B);          // enciende MPU
dev8_write8(…)                                       // calibrar etc

ax = dev8_read16(0,MPU6050_RA_ACCEL_XOUT_H);         // lee aceleración X …
```

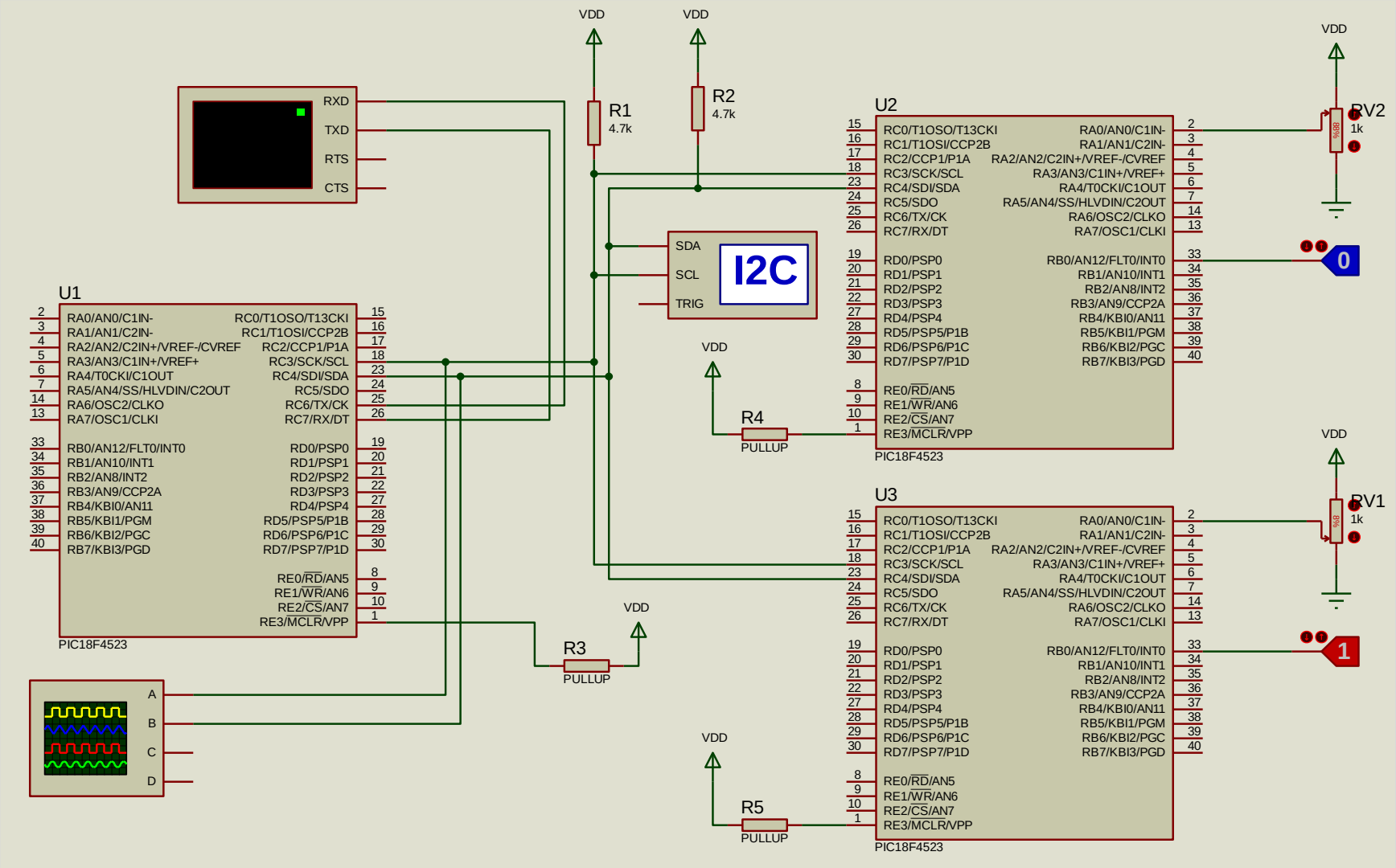
Ejemplo 01 en PIC16F884: I2C_16F884_24LC256

Accede a dos EEPROM 24LC256, utilizando la biblioteca mi24LC256.c para las operaciones de escritura y lectura aleatoria y secuencial vistas. El ensayo se hace a través del terminal serie.



Ejemplo 02 en PIC 18F4523: I2C_Master_Slave_18F4523

Un µC Maestro Accede a dos µC Esclavos I2C, solicitando lecturas de valores (entradas analógicas)



A/D

Conversor Analógico Digital (A/D)

- Este modulo del µC permite la conversión de un voltaje en un cierto rango analógico (por ejemplo 0 a 5 volts) en una representación binaria proporcional, con una cantidad de bits que va típicamente de 8 a 24. La arquitecturas electrónicas para realizar esta conversión son muy variadas, las más rápidas (GigaS/s, pero usualmente de menor resolución, 8 bits) son *flash ADC* y *pipeline ADC,* las de mayor resolución (24 bits, pero más lentas, 1kS/s) son *sigma-delta* y *doble rampa.*

- En µC se utilizan conversores A/D de Aproximaciones Sucesivas, con una resolución de 10 ó 12 bits, aunque hay µCs especializados con A/Ds *Sigma-Delta* de hasta 24 bits. Las tasas de muestreo sobre un mismo canal van de unos 50kS/s en los microcontroladores de gama baja-media hasta unos 7MS/s en microcontroladores de gama alta.

- Los µC disponen de varios canales analógicos que son seleccionados mediante un multiplexor interno para su conversión en un único A/D. En µCs de gama media-alta puede haber más conversores A/D, sea para un muestreo simultáneo de señales o para entrelazar las conversiones sobre una misma señal para obtener una mayor tasa de muestreo.

- Como en la mayoría de los periféricos, las entradas para el subsistema A/D son compartidas con otras posibles funciones de pin.

Consideraciones de selección y uso de un A/D

Consideraciones previas para seleccionar microcontrolador y/o A/D externo

1) Ancho de banda de la señal → frecuencia mínima de muestreo. ¿Alcanza la f del µC?
2) Rango dinámico de la señal → Número de bits del A/D. ¿Alcanza la resolución del µC?

2) ¿Entrada simple o diferencial?. La mayoría de los micros sólo admite entrada analógica simple. Algunos incorporan amplificadores operacionales para armar un amplificador diferencial. La opción es usar un amplificador operacional o de instrumentación externo, o un A/D externo de entrada diferencial (ejemplo MCP3553, A/D de 22 bits por SPI)

Elementos básicos a configurar inicialmente en subsistema A/D:

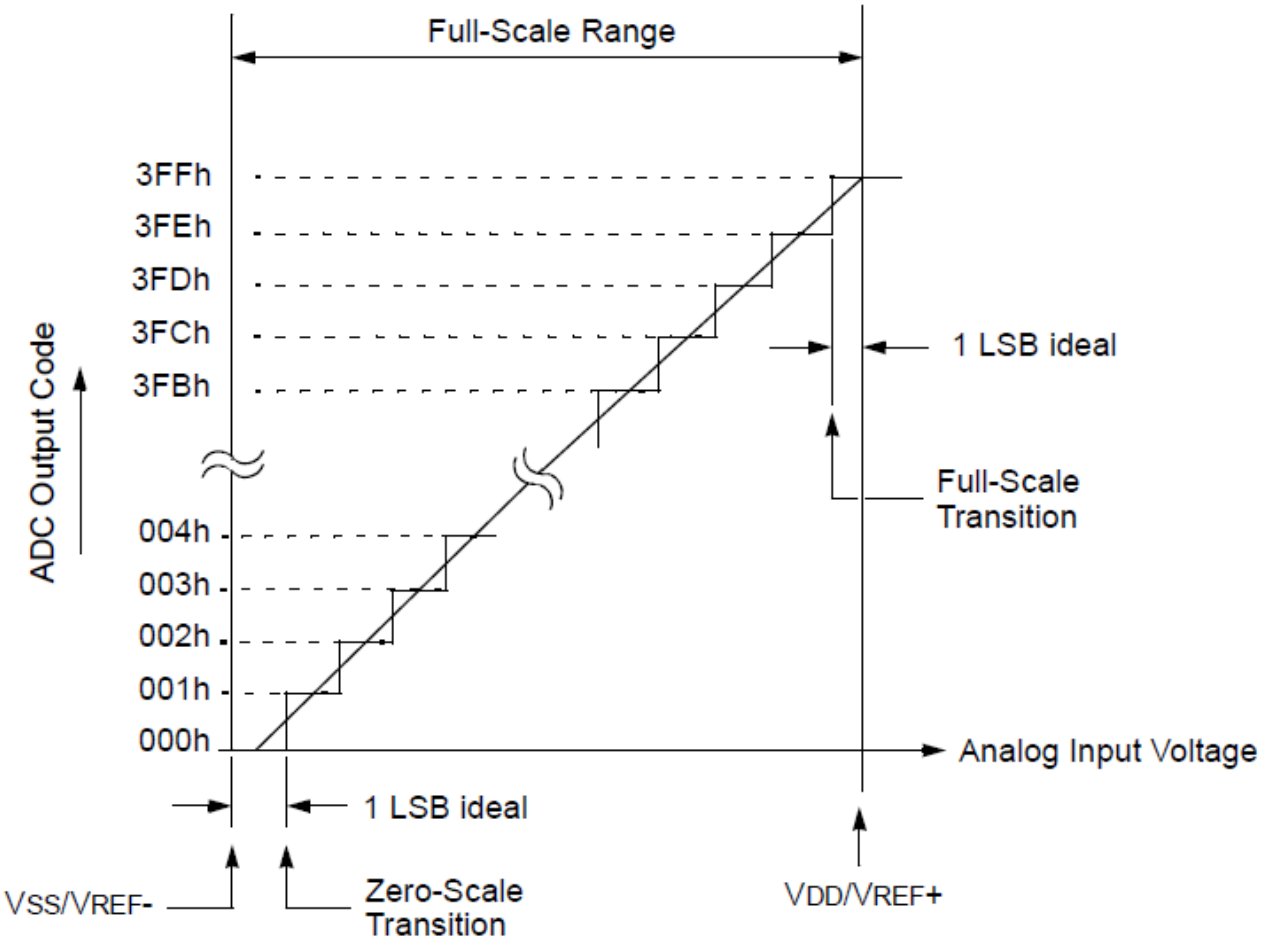
- Pines destinados a entradas analógicas.
- Pines destinados a +Vref y –Vref (opcional, puede usarse Vdd-Vss, ó Avdd-Vss)
- Velocidad de conversión. Elección del reloj (derivado del sistema Fcy) o de un oscilador RC propio.
- Cantidad de bits de conversión: 1 byte (8 bits) ó 2 bytes (típico 10 o 12 bits total).
- Alineamiento de la conversión. A izquierda (MSB es b15) o a derecha (LSB es b0)
- Uso de interrupción por Fin de Conversión
- Si se requiere muestreo uniforme, utilizar una fuente de muestreo uniforme (Ej. Timer)
- Uso de DMA (sólo en micros de gama alta, con módulo DMA).

En cada conversión:

Si se utilizan múltiples canales, deberá realizarse

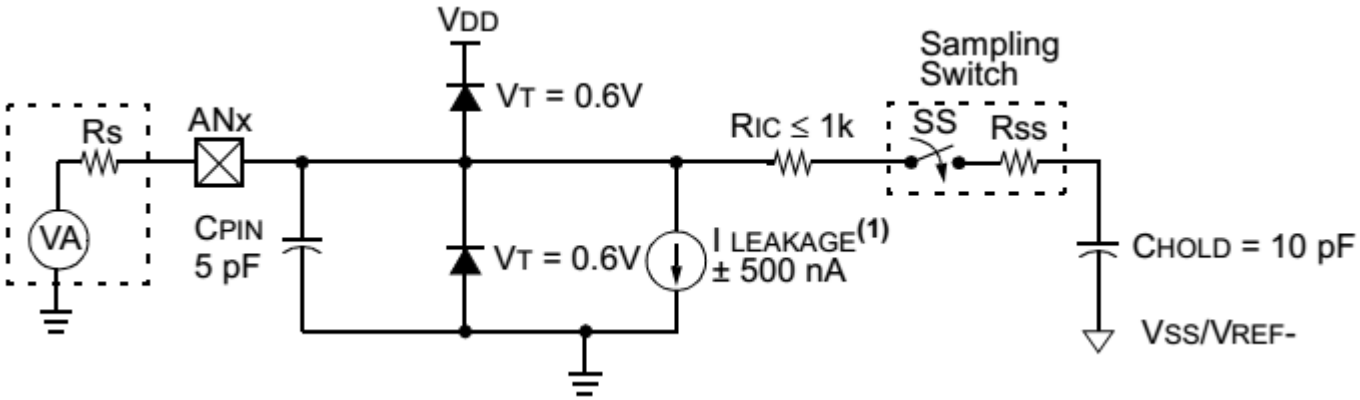
- selección del canal
- espera según impedancias en juego (de pocos µs a algunos ms)
- inicio de conversión
- *Polling* de fin de conversión o por interrupción

Función de transferencia (ideal) de A/D



El rango de entrada del A/D está entre los voltajes Vref(-) y Vref(+). Habitualmente se utiliza Vref(-)=0 y Vref(+)=Vdd, pero el micro permite también establecerlas externamente.
En un A/D de 10 bits, la función de transferencia va de 0h para Vref(-), hasta 3FFh (1023) para Vref(+).
El tamaño de un escalón (1 LSB) será [Vref(+)-Vref(-)]/1023.
Por ejemplo para Vref(+)=5 y Vref(-)=0 es 1 LSB = 5/1023 ≈ 5mV

Espera entre cambio de canal e inicio de conversión

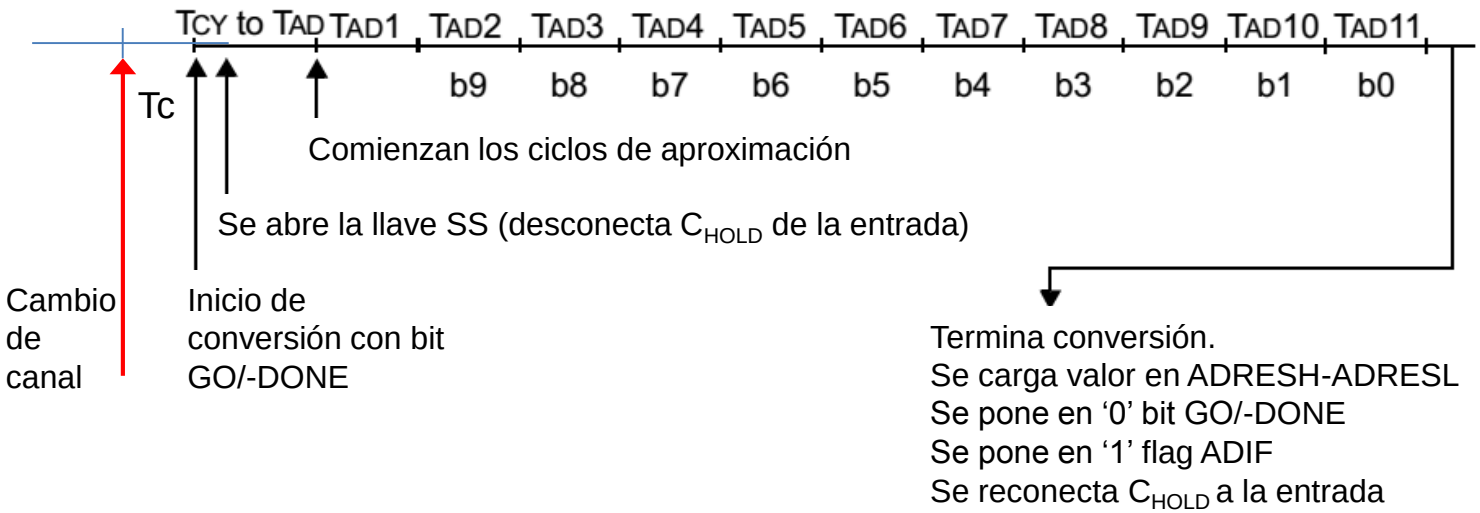


Modelo de entrada analógica (PIC16F88x, similar a ATmega).
El conversor A/D más habitual en microcontroladores es el de aproximaciones sucesivas, y necesita un seguidor-retenedor S&H para mantener el voltaje mientras se convierte.
Cuando la llave de muestreo SS se cierra, la tensión de entrada VA queda aplicada al capacitor C_{HOLD} a través de las resistencias Rs, Ric y Rss.
El caso más desfavorable ocurre cuando $|VA - V_{HOLD}|$ es máxima, esto es del tamaño del rango, por ejemplo 5volts. Deberá transcurrir un tiempo Tc hasta que V_{HOLD} alcance a VA con un error de ½ LSB, que se puede estimar como:

$Tc = (Rs+Ric+Rss).Chold.\ ln(2047)$

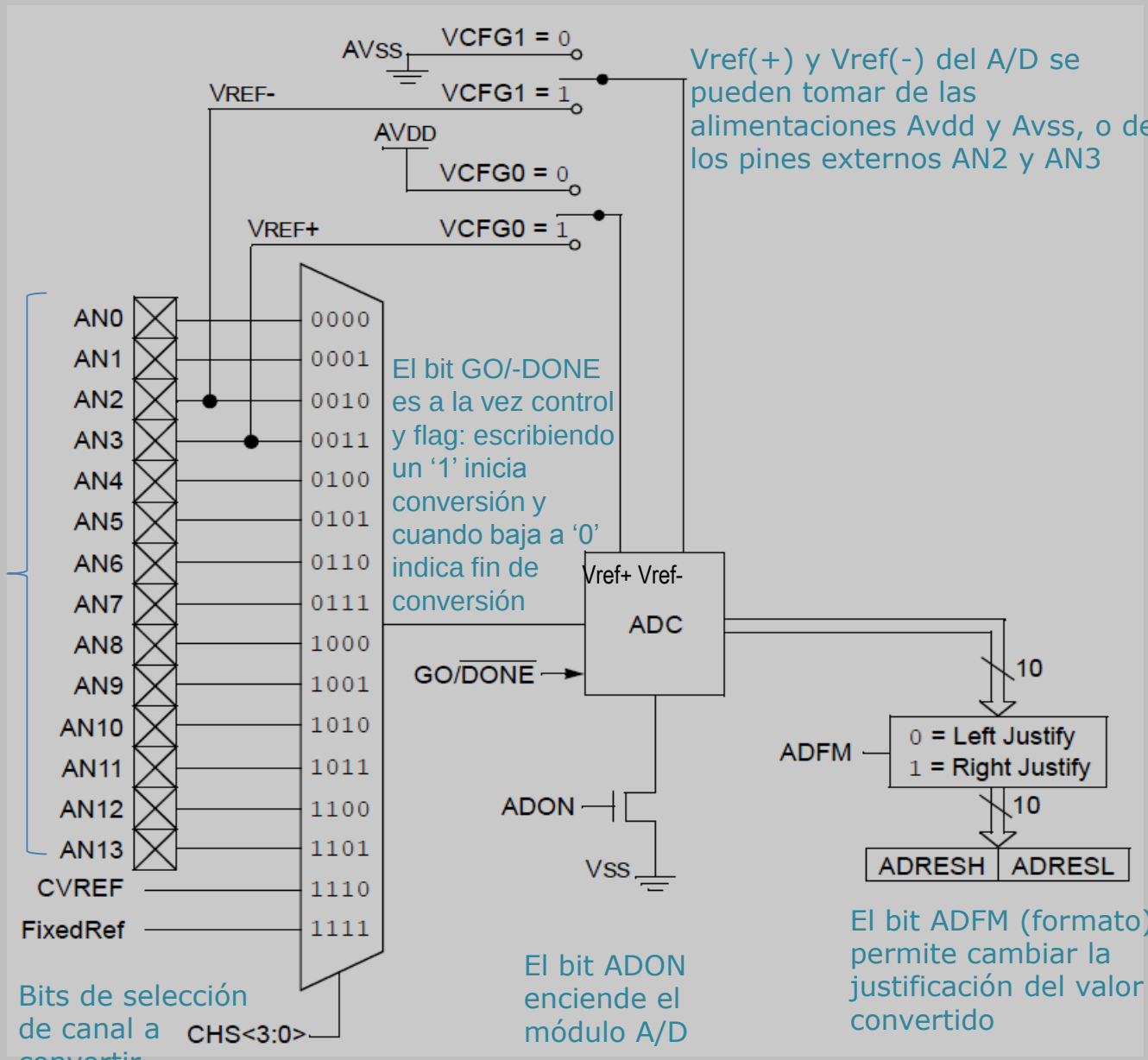
Ric y Rss son propios del micro y suman unos 8KΩ, Chold también y es de unos10pf.
Rs es la impedancia de salida de la fuente a convertir, que debería ser menor que 10KΩ.
Con estos valores, resulta. Tc = 1,37us.
A este tiempo hay que agregar unos 2us y 0,05us/°C desde 25°C (por ejemplo a 55 grados son unos 1,5us más).
Es decir, hay unos 4 o 5 us que debe esperarse entre un cambio de canal y el inicio de una conversión.

Tiempo de conversión (PIC)



El tiempo periodo de las aproximaciones sucesivas TAD se determina seleccionando la fuente de clock con los bits ADCS1-ADCS0. No puede ser inferior a 1,6us. Es decir, la conversión completa demanda como mínimo para 10 bits unos 16us, a lo que debe sumarse los 4 o 5 us explicados anteriormente.
Con estos valores, el tiempo para obtener una muestra es Ts ≈ 20 us (Fs ≈ 50kS/s)
Usando el oscilador RC interno es T_{AD} ≈ 4 us, dando un Ts ≈ 44 us (Fs ≈ 27kS/s)

Subsistema A/D del 16F884



Vref(+) y Vref(-) del A/D se pueden tomar de las alimentaciones Avdd y Avss, o de los pines externos AN2 y AN3

Entradas Analógicas (ver pines 2,3, etc)

El bit GO/-DONE es a la vez control y flag: escribiendo un '1' inicia conversión y cuando baja a '0' indica fin de conversión

Bits de selección de canal a convertir

El bit ADON enciende el módulo A/D

El bit ADFM (formato) permite cambiar la justificación del valor convertido

Registros del subsistema A/D del 16f884: ADCON0

ADCON0: A/D CONTROL REGISTER 0

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

	Tipo (*)	Nombre	función
7	R/W-0	ADCS1	
6		ADCS0	

ADC Clock Period (TAD)		Device Frequency (FOSC)			
ADC Clock Source	ADCS<1:0>	20 MHz	8 MHz	4 MHz	1 MHz
FOSC/2	00	100 ns(2)	250 ns(2)	500 ns(2)	2.0 µs
FOSC/8	01	400 ns(2)	1.0 µs(2)	2.0 µs	8.0 µs(3)
FOSC/32	10	1.6 µs	4.0 µs	8.0 µs(3)	32.0 µs(3)
FRC	11	2-6 µs(1,4)	2-6 µs(1,4)	2-6 µs(1,4)	2-6 µs(1,4)

5	R/W-0	CHS3	Selección de canal del multiplexor analógico 0000 a 1101: Canales AN0 a AN13 1110: Voltaje de referencia Cvref, proporcional a Vdd 1111: Voltaje de referencia fijo de 0.6 volts
4		CHS2	
3		CHS1	
2		CHS0	
1	R/W-0	GO/-DONE	Se debe poner en ‘1’ para iniciar conversión. Cuando termina conversión se pone en ‘0’ (simultáneamente se activa flag ADIF)
0	R/W/-0	ADON	En ‘1’ activa módulo A/D, en ‘0’ el A/D está desactivado y no consume corriente

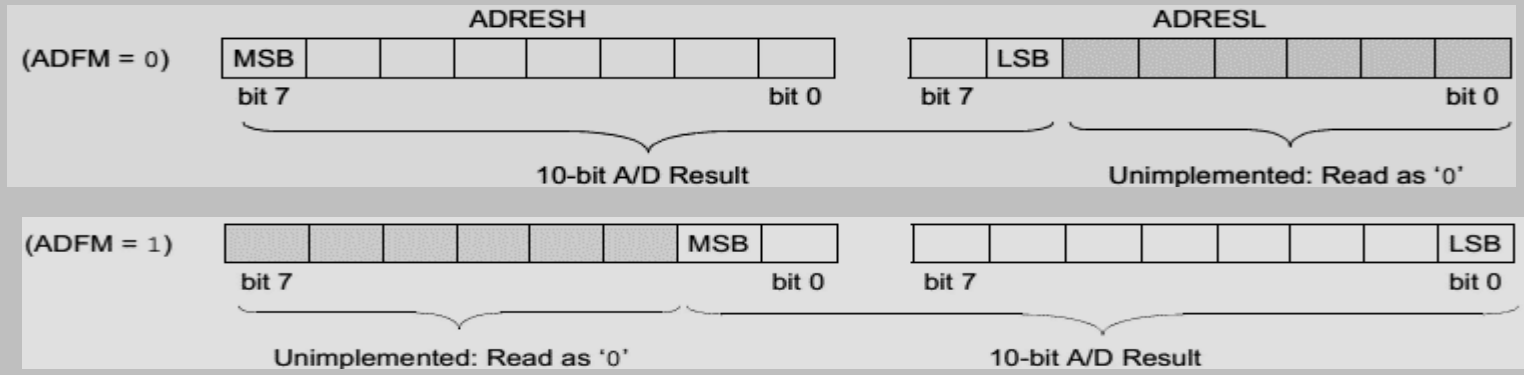
Registros del subsistema A/D del 16f884: ADCON1, ADRESx, ANSELx

ADCON1: A/D CONTROL REGISTER 1

R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0
ADFM	—	VCFG1	VCFG0	—	—	—	—
bit 7							bit 0

ADFM: Con 0 justifica a izquierda (0 a 1111111111000000b), con 1 a derecha (0 a 1023)
VCFG0=0 → Vref(+) = Vdd (típ 5 volts) , VCFG1=1 →Vref(+)=AN3
VCFG1=0 → Vref(-) = Vss (0 volts), VCFG1=1 →Vref(-)=AN2

ADRESL y ADRESH: Son los registros en los que se almacena el valor convertido



ANSEL y ANSELH: Son los registros para configurar pines como entradas analógicas

Para habilitar los pines como entradas analógicas AN0 a AN13, se deben poner en '1' los correspondientes bits ANS0 a ANS13 de los registros ANSEL y ANSELH.

Registros asociados al A/D del 16f884 - resumen

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
ADCON0	ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON	0000 0000	0000 0000
ADCON1	ADFM	—	VCFG1	VCFG0	—	—	—	—	0-00 ----	-000 ----
ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0	1111 1111	1111 1111
ANSELH	—	—	ANS13	ANS12	ANS11	ANS10	ANS9	ANS8	--11 1111	--11 1111
ADRESH	A/D Result Register High Byte								xxxx xxxx	uuuu uuuu
ADRESL	A/D Result Register Low Byte								xxxx xxxx	uuuu uuuu
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000x
PIE1	—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	-000 0000	-000 0000
PIR1	—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	-000 0000	-000 0000
PORTA	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0	xxxx xxxx	uuuu uuuu
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx	uuuu uuuu
PORTE	—	—	—	—	RE3	RE2	RE1	RE0	---- xxxx	---- uuuu
TRISA	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	1111 1111	1111 1111
TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111	1111 111
TRISE	—	—	—	—	TRISE3	TRISE2	TRISE1	TRISE0	---- 1111	---- 111

Pasos para usar el subsistema A/D

Configuración inicial (1 vez)
1. Configurar pin como entrada (registro TRIS) y como analógico (registro ANSEL en el 16F88x, ver en otros)
2. Seleccionar clock de ADC (bits ADCS), Vref (bits VCFG), formato de resultado (bit ADFM).
3. Encender módulo (bit ADON).
4. Si se va a habilitar interrupción por fin de conversión, borrar flag de interrupción ADIF (porque puede encenderse accidentalmente) y habilitar ADIE.

En cada muestra.
5. Seleccionar canal de entrada (bits CHSn).
6. Esperar estabilización de la señal. T_{AD}
7. Iniciar conversión con '1' en bit GO/-DONE
8. Esperar que el bit GO/-DONE pase a '0' o ADIF a '1' (polling),
9. Leer resultado (registros ADRESH y ADRESL)
10. Borrar ADIF (si fuera utilizada la interrupción o el polling de ADIF)

En caso de usar un solo canal, el paso 5 se puede hacer por única vez. También la estabilización es más rápida.

Funciones del CCS para usar el subsistema A/D

Configurar pines de puertos como entradas analógicas

```
setup_adc_ports(sAN0|sAN1|sAN2|sAN3);
```

Equivale a ANSEL = 0b00001111

Seleccionar reloj para la conversión

```
setup_adc(ADC_CLOCK_DIV_8);
```

Equivale a ADCON0.ADCS10 = 0b01

Seleccionar formato

```
#device adc=10  (directiva al compilador)
```

Equivale a ADCON1.ADFM = 1

Seleccionar canal

```
set_adc_channel(n);
```

Equivale a ADCON0.CHS30 = n

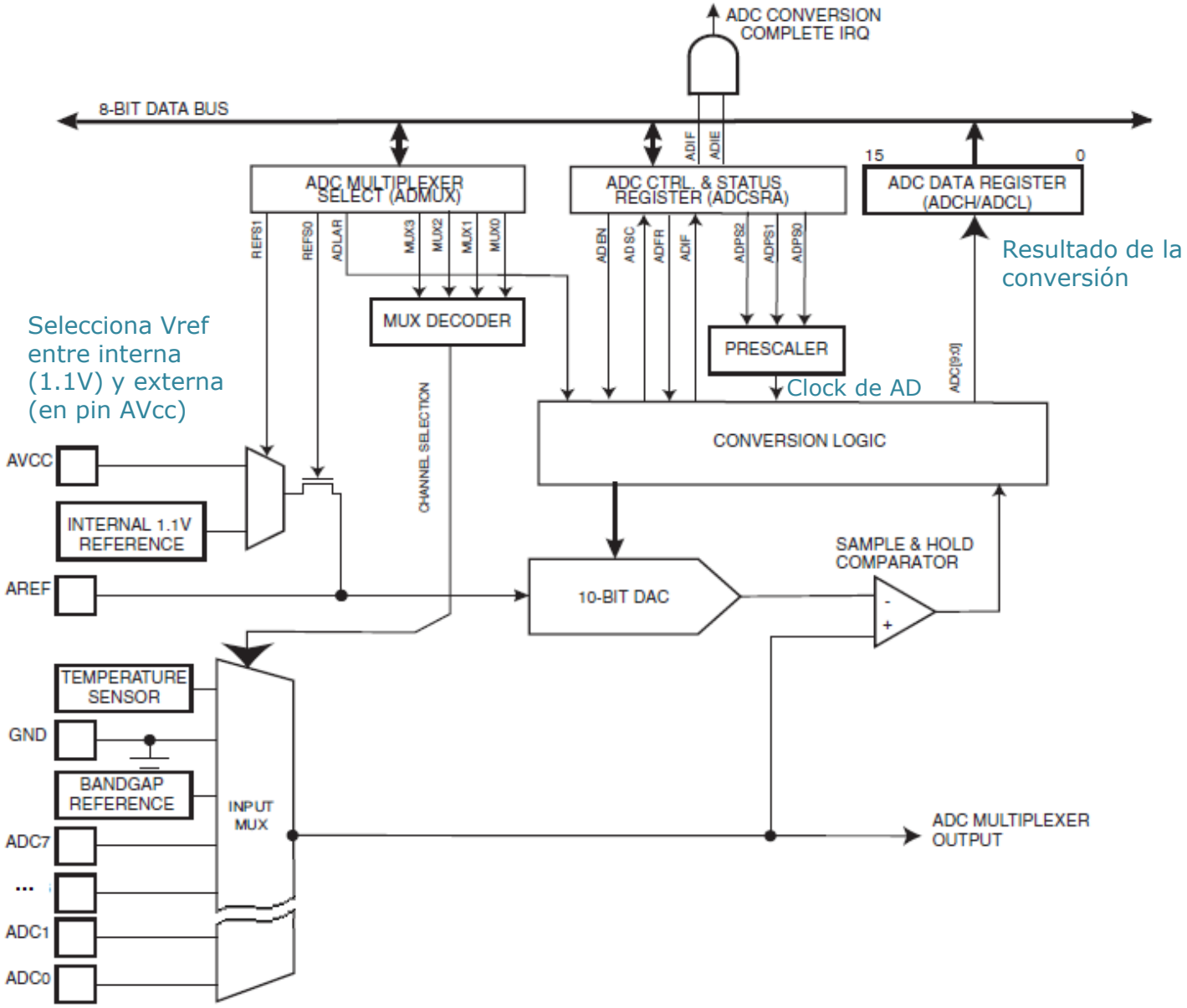
Leer ADC y guardar en variable "lectura"

```
lectura=read_adc();
```

Equivale a

```
ADCON0.GO_DONE=1;
while(ADCON0.GO_DONE);
lectura=((unsigned int16)ADRESH)<<8|ADRESL;
```

Subsistema A/D del ATmega328P



Registros del subsistema A/D del ATmegaxxx

Registro ADMUX

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

0 0 AREF
0 1 Avcc
1 0 reservada
1 1 Internal 1.1V

Alinear a izquierda

Canal
0 a 7: entrada ANA
8: Sensor Temp
9 a 13: reservado
14: 1.1 (Vref)
15: GND

Registro de Datos

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

Registros del subsistema A/D del ATmegaxxx (2)

Registro Control y Estado A

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

Habilita ADC	Inicia conversión	Habilita Auto-Trigger	Flag de Fin de conversión	Habilita INT de Fin de conversión	Prescaler de clock de ADC 000: 2, 001: 2, 010: 4, 011: 8 100:16,101:32, 010:64,111: 128

Registro Control y Estado B

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

Bit para módulo comparador

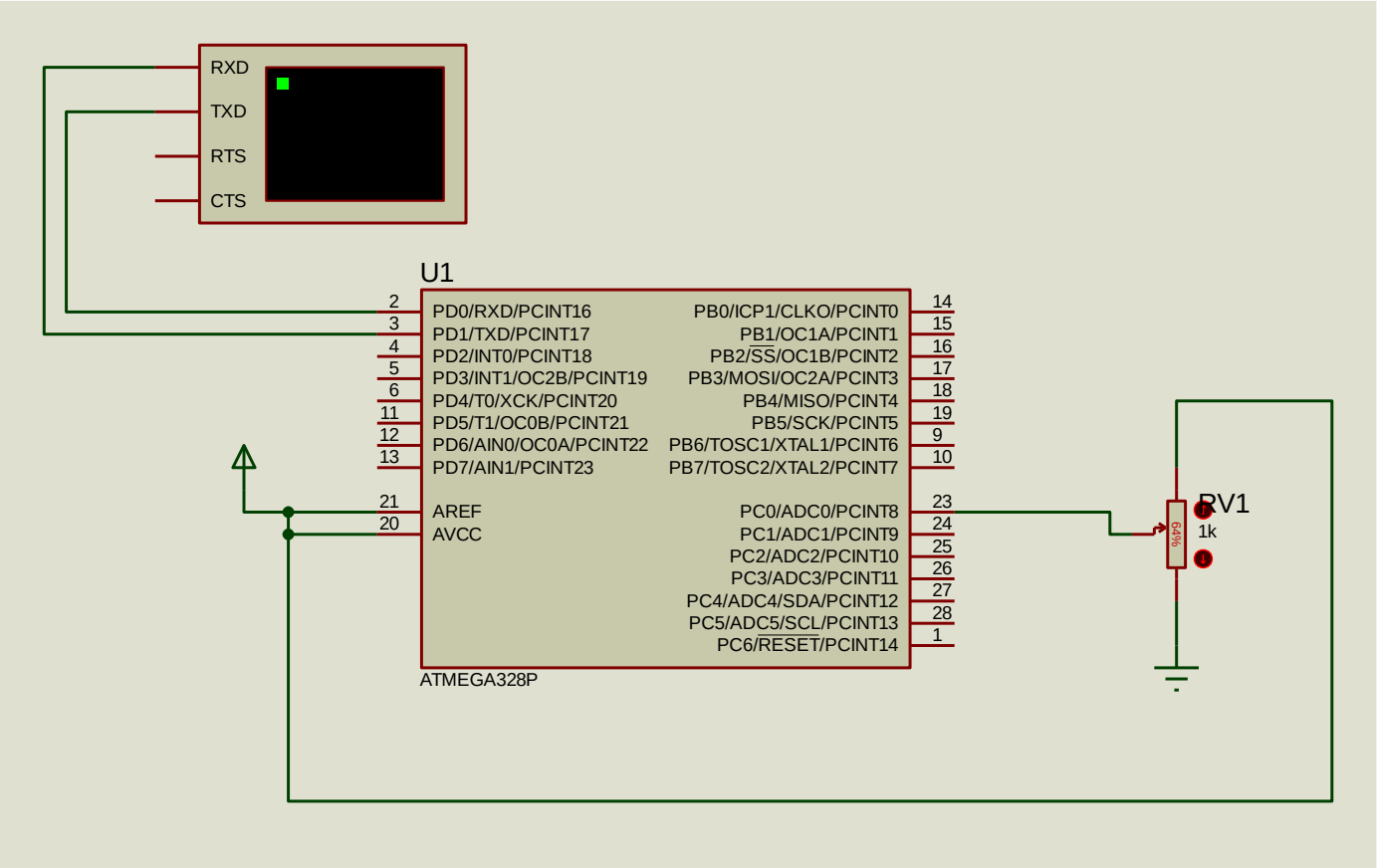
Selección de Auto-trigger
000: free-running (continuo)
001: Comparador analógico
010: INT0
011: TIMER0 compare match A
100: TIMER0 overflow
101: TIMER1 compare match B
110: TIMER1 overflow
111: Timer1 capture event

Registro deshabilitador de DIs

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	

Escribiendo 1 en los bits ADC0D a ADC5D, se deshabilitan los buffers de entrada digital (que cuando son **analógicas** no se necesitan). Así se reduce el consumo de la entrada

Ejemplos. Conversor A/D en ATmega328



EJ01_ADC_Tx Implementación básica de un ADC en ATmega328
EJ02_ADC_Tx_estructura_parámetros: Implementa un ADC en ATmega328 con estructura de datos para la configuración del AD
EJ03_ADC_Tx_MuestreoUniforme: Utiliza un timer para realizar muestreo periódico, e interrupción.
Como el ejemplo anterior, implementa un ADC en ATmega328 con estructura de datos para la configuración del AD.

Microcontroladores con núcleo ARM

Microprocesadores ARM

- Microprocesadores RISC (Advanced RISC Machine) de 32 y 64 bits arquitectura RISC (Reduced Instruction Set Computer) y Harvard Modificada (separación física de datos e instrucciones, pero con posibilidad de tratar instrucciones como datos).
- Núcleos licenciables.

- Instrucciones de ejecución condicional y transferencia entre distintos registros en un solo ciclo, lo que permite obtener código muy eficiente (compacto y rápido).

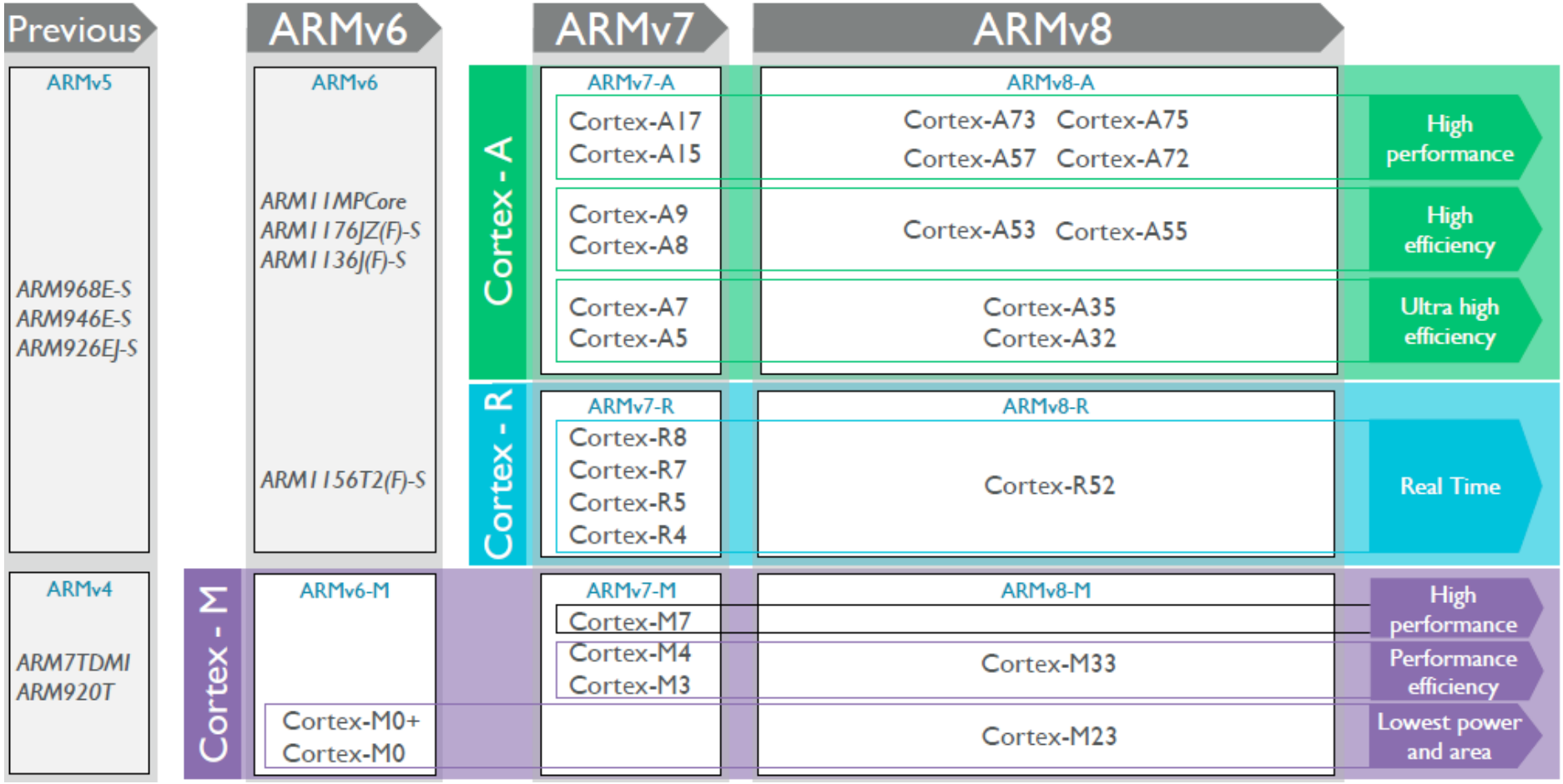
- Gran capacidad de direccionamiento (4GB en los de 32 bits)

- Registros de periféricos mapeados en memoria (MMIO: Memory mapped I/O), con amplia franja de direcciones por periférico, lo que se utiliza para alojar registros de control permitiendo gran flexibilidad de configuraciones (funciones alternativas en pines, tipo de entrada y salida etc)

- Dos estados operativos: ARM (instrucciones 32 bits) y Thumb (16 bits)

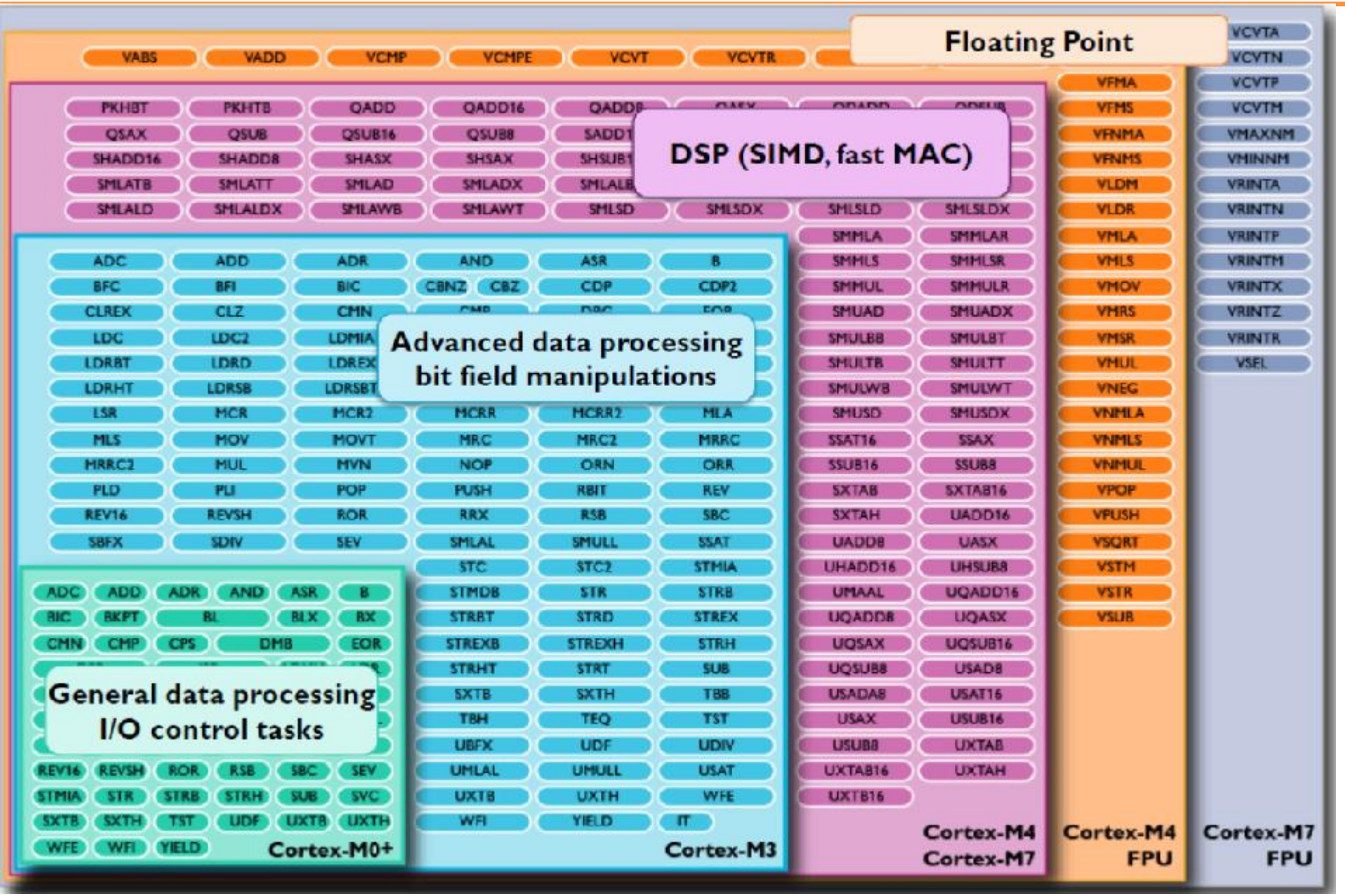
- Cada fabricante incorpora sus propios periféricos, con sus propios registros de configuración y uso, pero se respetan las regiones de memoria.

Familia de procesadores ARM

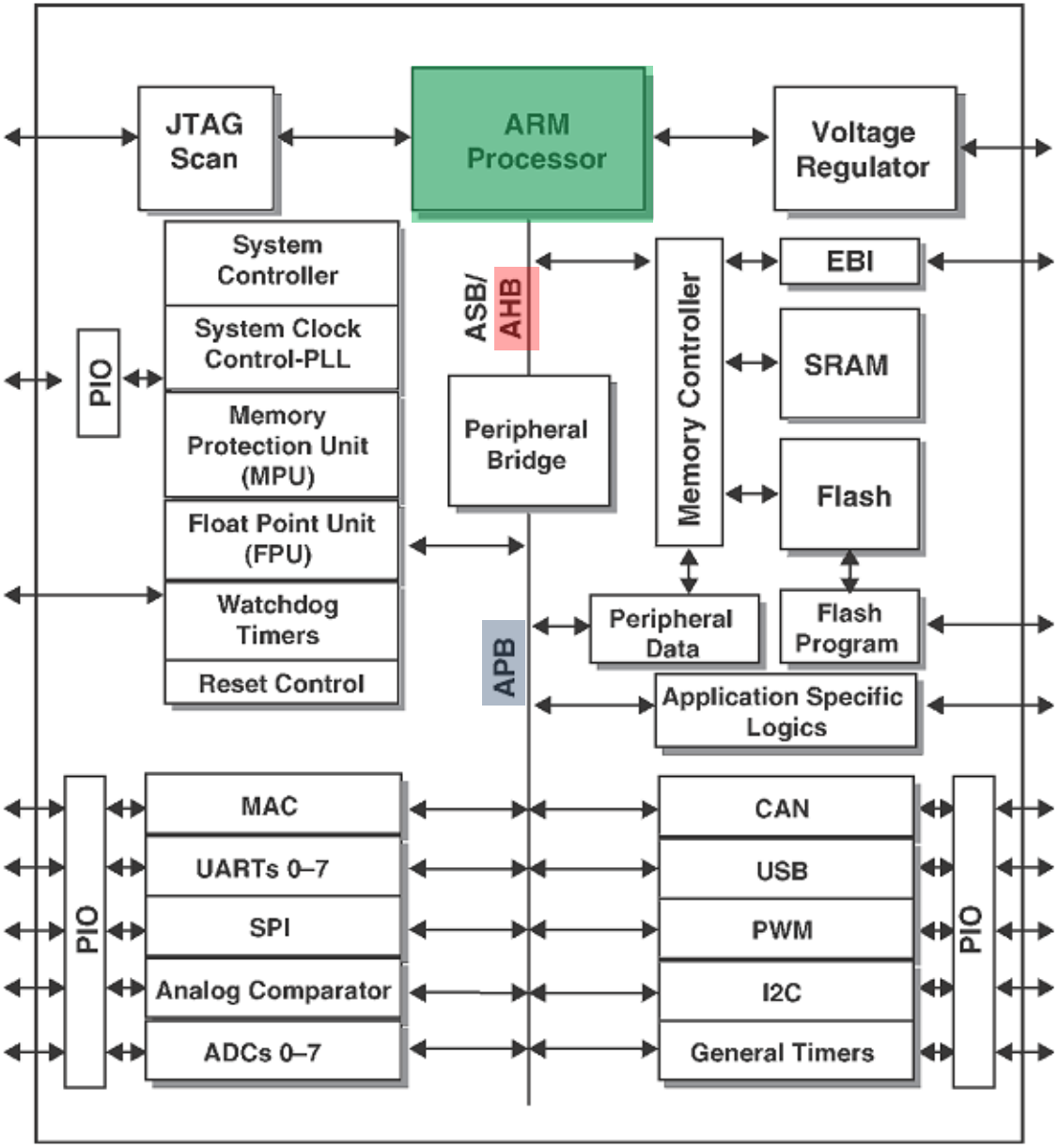


http://infocenter.arm.com

Cortex-M. Set de instrucciones - compatibilidad



Interconexión de sistemas en un Cortex-M4 típico



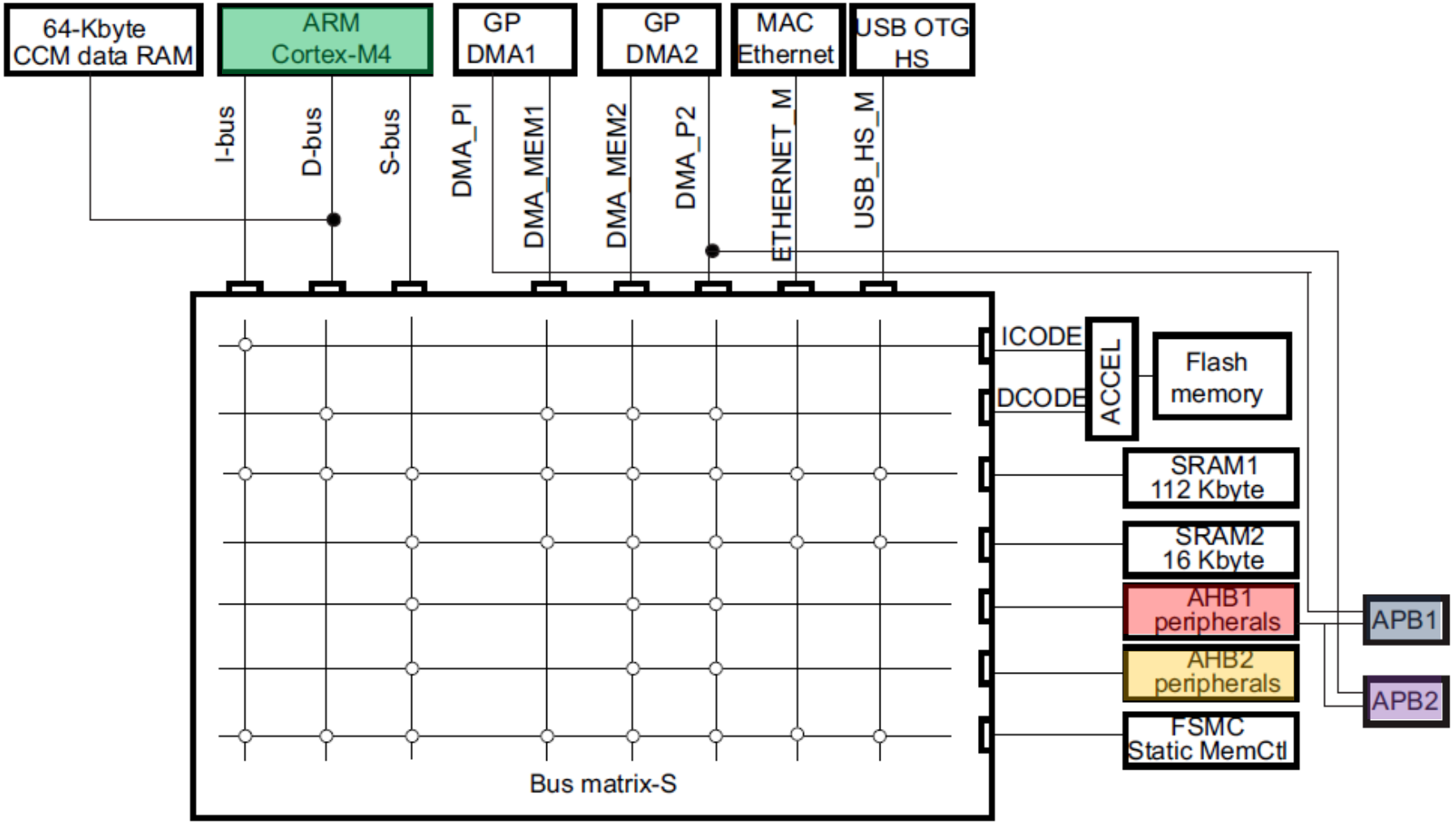
AMBA:
Advanced Microcontroller Bus Architecture

AMBA AHB:
Advanced High-performance Bus
- Alto rendimiento
- Operación pipelined
- Múltiples maestros
- Transferencia en ráfaga (burst)
- Transacciones partidas

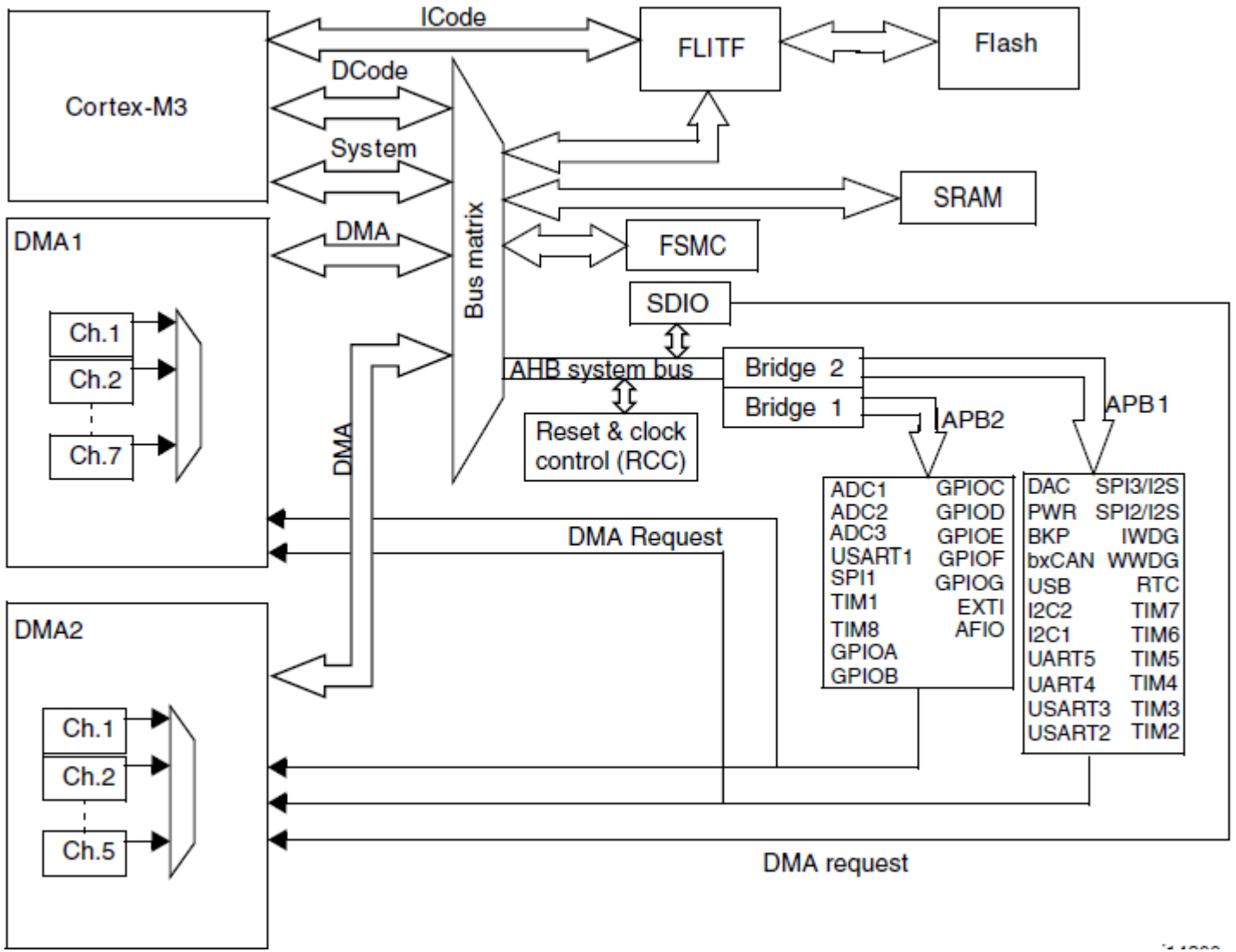
AMBA ASB:
Advanced System Bus
- Alto rendimiento
- Operación pipelined
- Múltiples maestros

AMBA APB:
- Advanced Peripheral Bus
- Bajo consumo
- *Latch* de Dirección y Control
- Interfaz simple

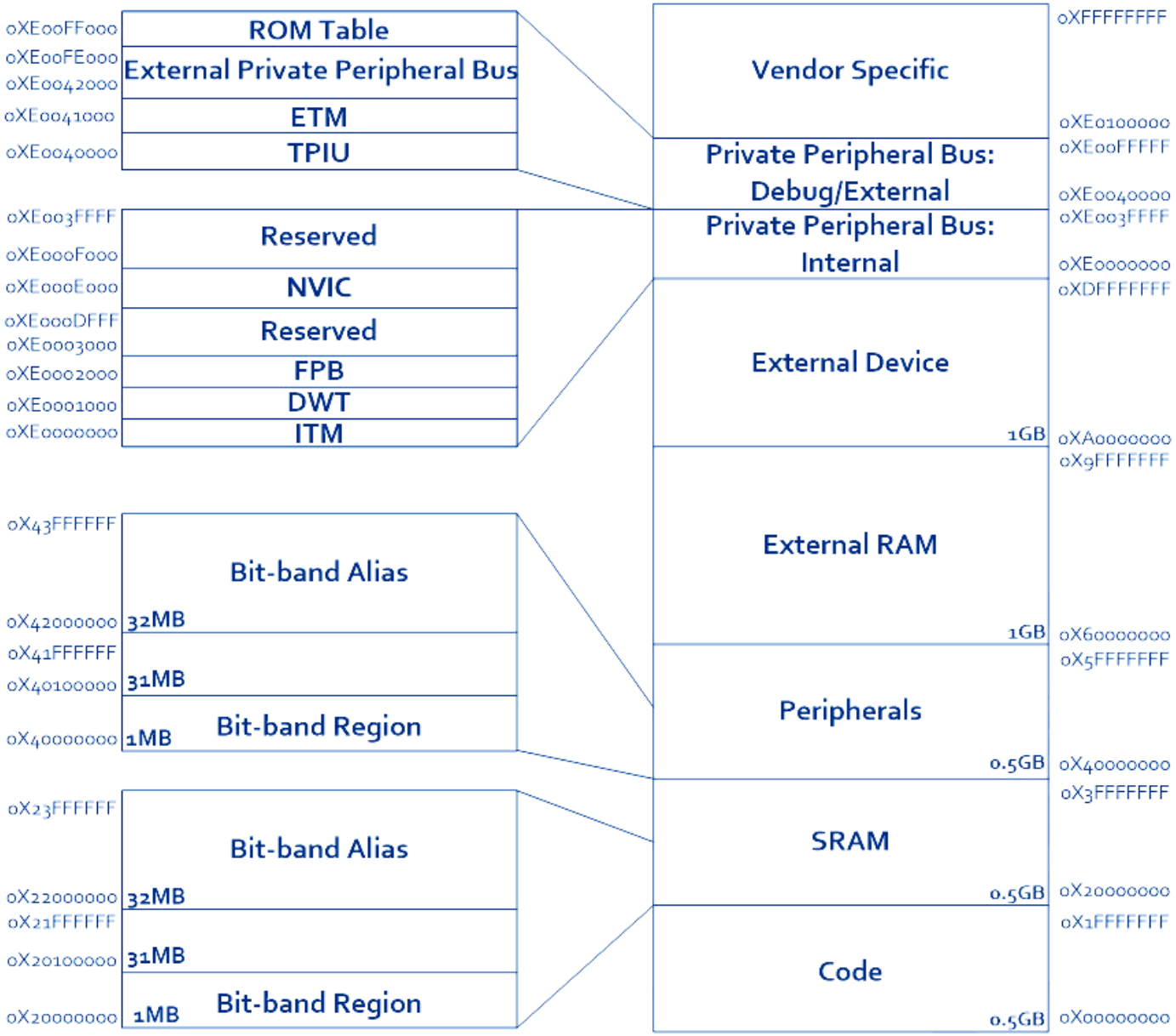
Interconexión de sistemas en un Cortex M4 (STM32f4xx)



Interconexión de sistemas en un Cortex-M3 típico



Mapa de memoria de Cortex M3 y M4



Direcciones base del STM32F4xx

Definiciones extraídas de stm32f4xx.h

```
#define FLASH_BASE    ((uint32_t)0x08000000) /*!< FLASH(up to 1 MB) base address in the alias region
#define SRAM1_BASE      ((uint32_t)0x20000000) /*!< SRAM1(112 KB) base address in the alias region
#define PERIPH_BASE     ((uint32_t)0x40000000) /*!< Peripheral base address in the alias region
#define SRAM1_BB_BASE   ((uint32_t)0x22000000) /*!< SRAM1(112 KB) base address in the bit-band region
#define PERIPH_BB_BASE  ((uint32_t)0x42000000) /*!< Peripheral base address in the bit-band region
#define BKPSRAM_BB_BASE ((uint32_t)0x42024000) /*!< Backup SRAM(4 KB) base address in the bit-band
region

/* Legacy defines */
#define SRAM_BASE             SRAM1_BASE
#define SRAM_BB_BASE          SRAM1_BB_BASE

/*!< Peripheral memory map */
#define APB1PERIPH_BASE       PERIPH_BASE
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x00010000)
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000)
#define AHB2PERIPH_BASE       (PERIPH_BASE + 0x10000000)
```

Periféricos del bus APB1 del STM32F4xx

Definiciones extraídas de stm32f4xx.h

```
#define TIM2_BASE               (APB1PERIPH_BASE + 0x0000)
#define TIM3_BASE               (APB1PERIPH_BASE + 0x0400)
#define TIM4_BASE               (APB1PERIPH_BASE + 0x0800)
#define TIM5_BASE               (APB1PERIPH_BASE + 0x0C00)
#define TIM6_BASE               (APB1PERIPH_BASE + 0x1000)
#define TIM7_BASE               (APB1PERIPH_BASE + 0x1400)
#define TIM12_BASE              (APB1PERIPH_BASE + 0x1800)
#define TIM13_BASE              (APB1PERIPH_BASE + 0x1C00)
#define TIM14_BASE              (APB1PERIPH_BASE + 0x2000)
#define RTC_BASE                (APB1PERIPH_BASE + 0x2800)
#define WWDG_BASE               (APB1PERIPH_BASE + 0x2C00)
#define IWDG_BASE               (APB1PERIPH_BASE + 0x3000)
#define I2S2ext_BASE            (APB1PERIPH_BASE + 0x3400)
#define SPI2_BASE               (APB1PERIPH_BASE + 0x3800)
#define SPI3_BASE               (APB1PERIPH_BASE + 0x3C00)
#define I2S3ext_BASE            (APB1PERIPH_BASE + 0x4000)
#define USART2_BASE             (APB1PERIPH_BASE + 0x4400)
#define USART3_BASE             (APB1PERIPH_BASE + 0x4800)
#define UART4_BASE              (APB1PERIPH_BASE + 0x4C00)
#define UART5_BASE              (APB1PERIPH_BASE + 0x5000)
#define I2C1_BASE               (APB1PERIPH_BASE + 0x5400)
#define I2C2_BASE               (APB1PERIPH_BASE + 0x5800)
#define I2C3_BASE               (APB1PERIPH_BASE + 0x5C00)
#define CAN1_BASE               (APB1PERIPH_BASE + 0x6400)
#define CAN2_BASE               (APB1PERIPH_BASE + 0x6800)
#define PWR_BASE                (APB1PERIPH_BASE + 0x7000)
#define DAC_BASE                (APB1PERIPH_BASE + 0x7400)
#define UART7_BASE              (APB1PERIPH_BASE + 0x7800)
#define UART8_BASE              (APB1PERIPH_BASE + 0x7C00)
```

IWDG:	Independent Watchdog
WWDG:	Window watchdog
I2S:	Inter-IC-Sound

Periféricos del bus APB2 del STM32F4xx

Definiciones extraídas de stm32f4xx.h

```
#define TIM1_BASE               (APB2PERIPH_BASE + 0x0000)
#define TIM8_BASE               (APB2PERIPH_BASE + 0x0400)
#define USART1_BASE             (APB2PERIPH_BASE + 0x1000)
#define USART6_BASE             (APB2PERIPH_BASE + 0x1400)
#define ADC1_BASE               (APB2PERIPH_BASE + 0x2000)
#define ADC2_BASE               (APB2PERIPH_BASE + 0x2100)
#define ADC3_BASE               (APB2PERIPH_BASE + 0x2200)
#define ADC_BASE                (APB2PERIPH_BASE + 0x2300)
#define SDIO_BASE               (APB2PERIPH_BASE + 0x2C00)
#define SPI1_BASE               (APB2PERIPH_BASE + 0x3000)
#define SPI4_BASE               (APB2PERIPH_BASE + 0x3400)
#define SYSCFG_BASE             (APB2PERIPH_BASE + 0x3800)
#define EXTI_BASE               (APB2PERIPH_BASE + 0x3C00)
#define TIM9_BASE               (APB2PERIPH_BASE + 0x4000)
#define TIM10_BASE              (APB2PERIPH_BASE + 0x4400)
#define TIM11_BASE              (APB2PERIPH_BASE + 0x4800)
#define SPI5_BASE               (APB2PERIPH_BASE + 0x5000)
#define SPI6_BASE               (APB2PERIPH_BASE + 0x5400)
```

Periféricos del bus AHB1 del STM32F4xx

Definiciones extraídas de stm32f4xx.h

```
#define GPIOA_BASE                (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE                (AHB1PERIPH_BASE + 0x0400)
#define GPIOC_BASE                (AHB1PERIPH_BASE + 0x0800)
#define GPIOD_BASE                (AHB1PERIPH_BASE + 0x0C00)
#define GPIOE_BASE                (AHB1PERIPH_BASE + 0x1000)
#define GPIOF_BASE                (AHB1PERIPH_BASE + 0x1400)
#define GPIOG_BASE                (AHB1PERIPH_BASE + 0x1800)
#define GPIOH_BASE                (AHB1PERIPH_BASE + 0x1C00)
#define GPIOI_BASE                (AHB1PERIPH_BASE + 0x2000)
```

Periféricos del bus AHB1 del STM32F4xx

Definiciones extraídas de stm32f4xx.h

```
#define CRC_BASE              (AHB1PERIPH_BASE + 0x3000)
#define RCC_BASE              (AHB1PERIPH_BASE + 0x3800)
#define FLASH_R_BASE          (AHB1PERIPH_BASE + 0x3C00)
#define DMA1_BASE             (AHB1PERIPH_BASE + 0x6000)
#define DMA1_Stream0_BASE     (DMA1_BASE + 0x010)
#define DMA1_Stream1_BASE     (DMA1_BASE + 0x028)
#define DMA1_Stream2_BASE     (DMA1_BASE + 0x040)
#define DMA1_Stream3_BASE     (DMA1_BASE + 0x058)
#define DMA1_Stream4_BASE     (DMA1_BASE + 0x070)
#define DMA1_Stream5_BASE     (DMA1_BASE + 0x088)
#define DMA1_Stream6_BASE     (DMA1_BASE + 0x0A0)
#define DMA1_Stream7_BASE     (DMA1_BASE + 0x0B8)
#define DMA2_BASE             (AHB1PERIPH_BASE + 0x6400)
#define DMA2_Stream0_BASE     (DMA2_BASE + 0x010)
#define DMA2_Stream1_BASE     (DMA2_BASE + 0x028)
#define DMA2_Stream2_BASE     (DMA2_BASE + 0x040)
#define DMA2_Stream3_BASE     (DMA2_BASE + 0x058)
#define DMA2_Stream4_BASE     (DMA2_BASE + 0x070)
#define DMA2_Stream5_BASE     (DMA2_BASE + 0x088)
#define DMA2_Stream6_BASE     (DMA2_BASE + 0x0A0)
#define DMA2_Stream7_BASE     (DMA2_BASE + 0x0B8)
#define ETH_BASE              (AHB1PERIPH_BASE + 0x8000)
#define ETH_MAC_BASE          (ETH_BASE)
#define ETH_MMC_BASE          (ETH_BASE + 0x0100)
#define ETH_PTP_BASE          (ETH_BASE + 0x0700)
#define ETH_DMA_BASE          (ETH_BASE + 0x1000)
```

CRC:	Unidad de cálculo de CRC
RCC:	Control de Clock y Reset
DMA:	Acceso directo a memoria
ETH:	Ethernet MAC

Periféricos del bus AHB2 del STM32F4xx

Definiciones extraídas de stm32f4xx.h

```
#define DCMI_BASE               (AHB2PERIPH_BASE + 0x50000)
#define CRYP_BASE               (AHB2PERIPH_BASE + 0x60000)
#define HASH_BASE               (AHB2PERIPH_BASE + 0x60400)
#define HASH_DIGEST_BASE        (AHB2PERIPH_BASE + 0x60710)
#define RNG_BASE                (AHB2PERIPH_BASE + 0x60800)
```

DCMI:	Interfaz de cámara digital
CRYP:	Procesador criptográfico
HASH:	Procesador de Hash
RNG:	Generador de números aleatorios

Herramientas de desarrollo

Toolchain GNU

https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

Última versión: gcc-arm-none-eabi-9-2019-q4-major-xx…

Otros compiladores propietarios. (ej. ARM compiler de ARM)

IDEs

Keil MDK-ARM:	De ARM-Keil. Comercial. Libre hasta 32k de código ejecutable.
Mbed:	De mbed.org. Version Cloud y offline. Permite exportar a otro IDE.
IAR	Embedded Workbench. De IAR. Comercial y libre limitado.
DS5-Development Studio.	De ARM. Basado en Eclipse. Versiones libres y comerciales.
CIAA.	Del proyecto-ciaa.com.ar. Eclipse personalizado. Libre. Por ahora sólo para CIAA-NXP
ChibiStudio.	De chibios.org. Eclipse personalizado. Con ChibiOS (rtos). Comercial y libre.
Crossworks for ARM.	De Rowley. Comercial y libre limitado.
TrueStudio.	De Atollic. Basado en Eclipse. Versiones libres y comerciales para varios fabricantes **hasta 8.x**. Desde 9.x de ST micro, versión full libre, pero sólo micros de ST. Discontinuado (ahora STMCubeIDE)
STMCubeIDE	De ST. Basado en Truestudio, que integra el STMCube para configuración gráfica. Solamente para micros de ST.
SW4STM.	De OpenSTM32.org. Libre. Colaborativo. Sólo para uC de ST

LPCOpen`para micros NXP. Code Composer Studio para micros Texas Instruments, Atmel Studio para micros Atmel etc.

Herramientas de desarrollo

Bibliotecas-HAL etc

CMSIS: Cortex Microcontroller Software Interface Standard. Ofrece una amplia HAL independiente del vendedor. Libre.

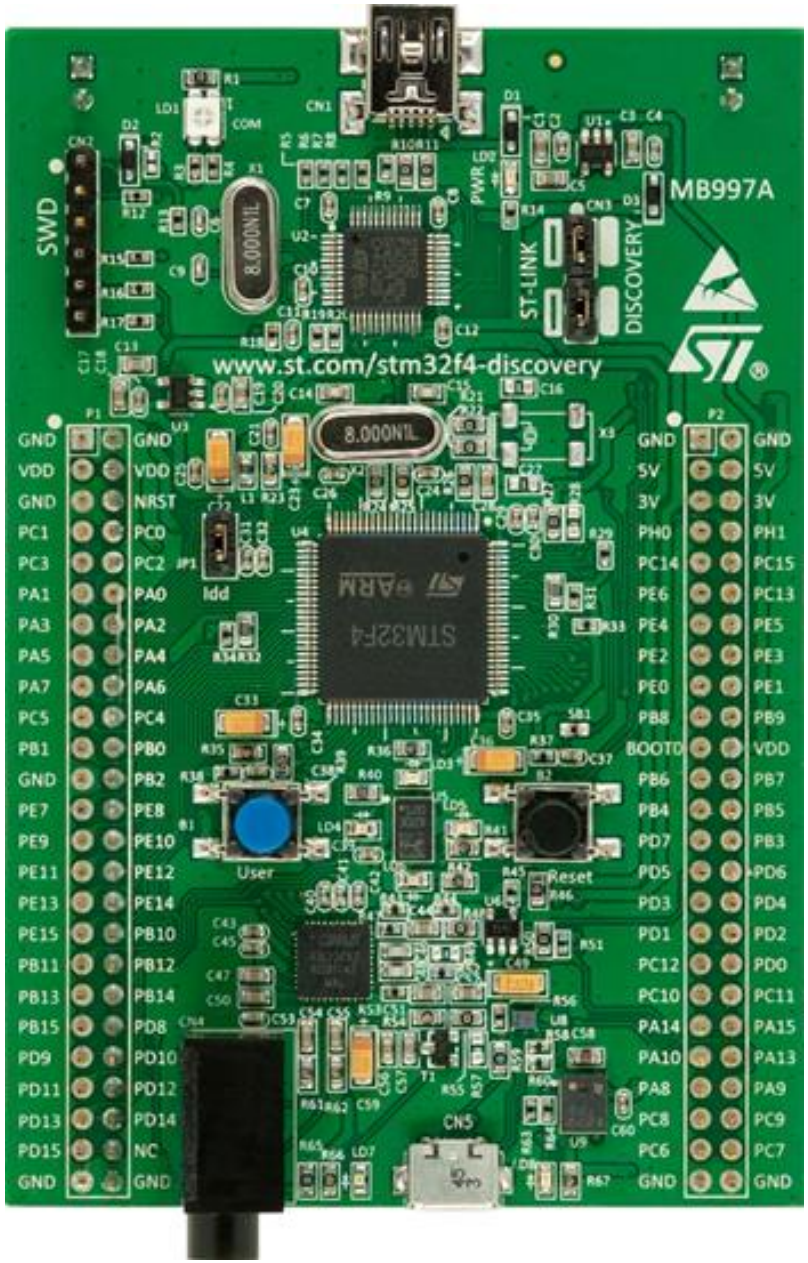
FreeRTOS, ChibiOS etc. Sistemas operativos que brindan RTOS y HAL.

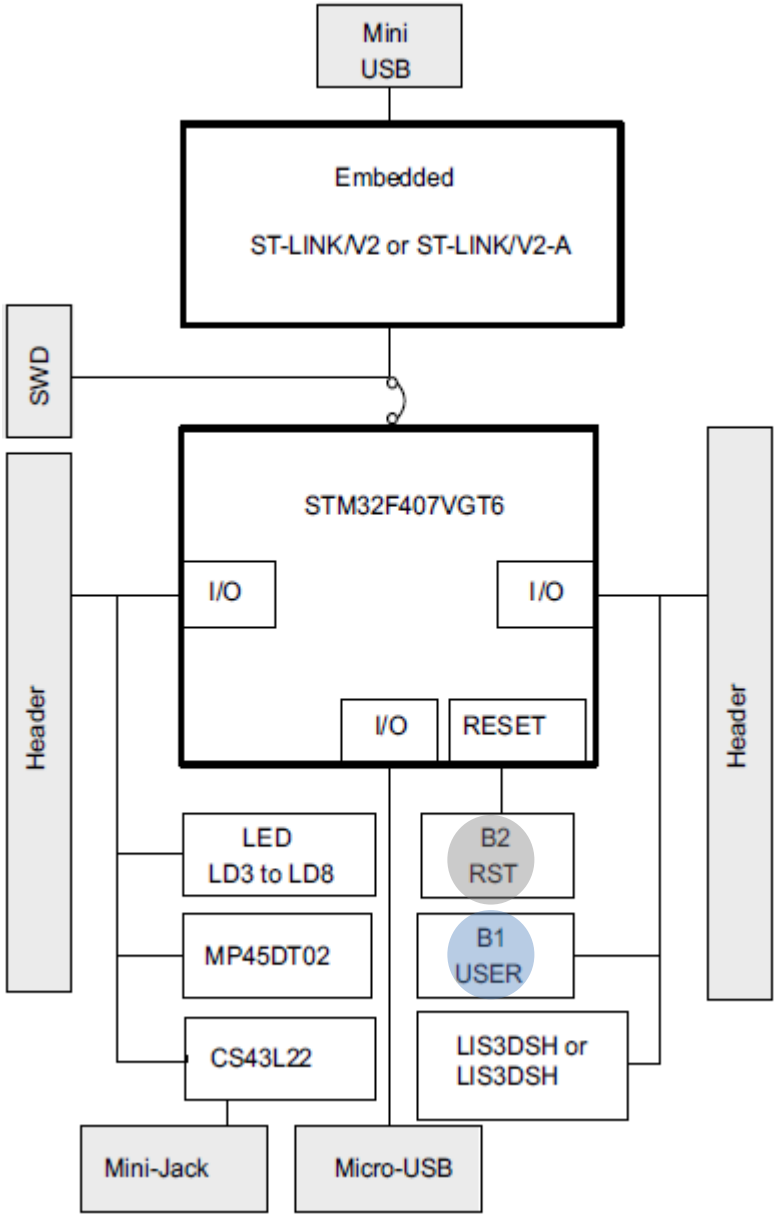
STMCube, CoSmart etc: Herramientas gráficas de configuración que brindan HAL.

Programadores, debuggers:

Hard JTAG SWD ST-Link. Soft GDB, OpenOCD etc

Prácticas sobre placa STM32F407 Discovery





Prácticas sobre placa STM32F407 Discovery

Ej00_STM32F407_MyEP_noH:	Blink básico con manejo directo de registros
EJ01_STM32F407_MyEP:	Blink básico utilizando cabecera stm32f4xx.h
Ej02_STM32F407_plantilla_TrueStudio:	Blink con asistente de TrueStudio
Ej03_STM32F407_MyEP_TIMER:	Blink con Timer
Ej04_STM32F407_MyEP_UART_básico:	Transmisión y recepción de bytes por UART.
Ej05_STM32F407_MyEP_UART_sprintf:	Transmisión de datos formateada con sprintf a UARTs
Ej06_STM32F407_MyEP_UART_INT:	Transmisión formateada de UART y recepción por Interrupción
EJ09_STM32F407_ADC:	Conversión A/D (con intervalo de timer) y transmisión por UART
EJ10_STM32F407_FreeRTOS_Blink:	Multitarea básico con FreeRTOS

Actualizado 8/6 hasta aquí

Subsistemas para Control de Movimiento

Entradas:

A/D

Input Capture

QEI

Se utilizan para determinar el estado de posición/velocidad/torque de un motor BDC (CC), BLDC (*Brushless*), AC asíncrono (*Inducción*) etc.

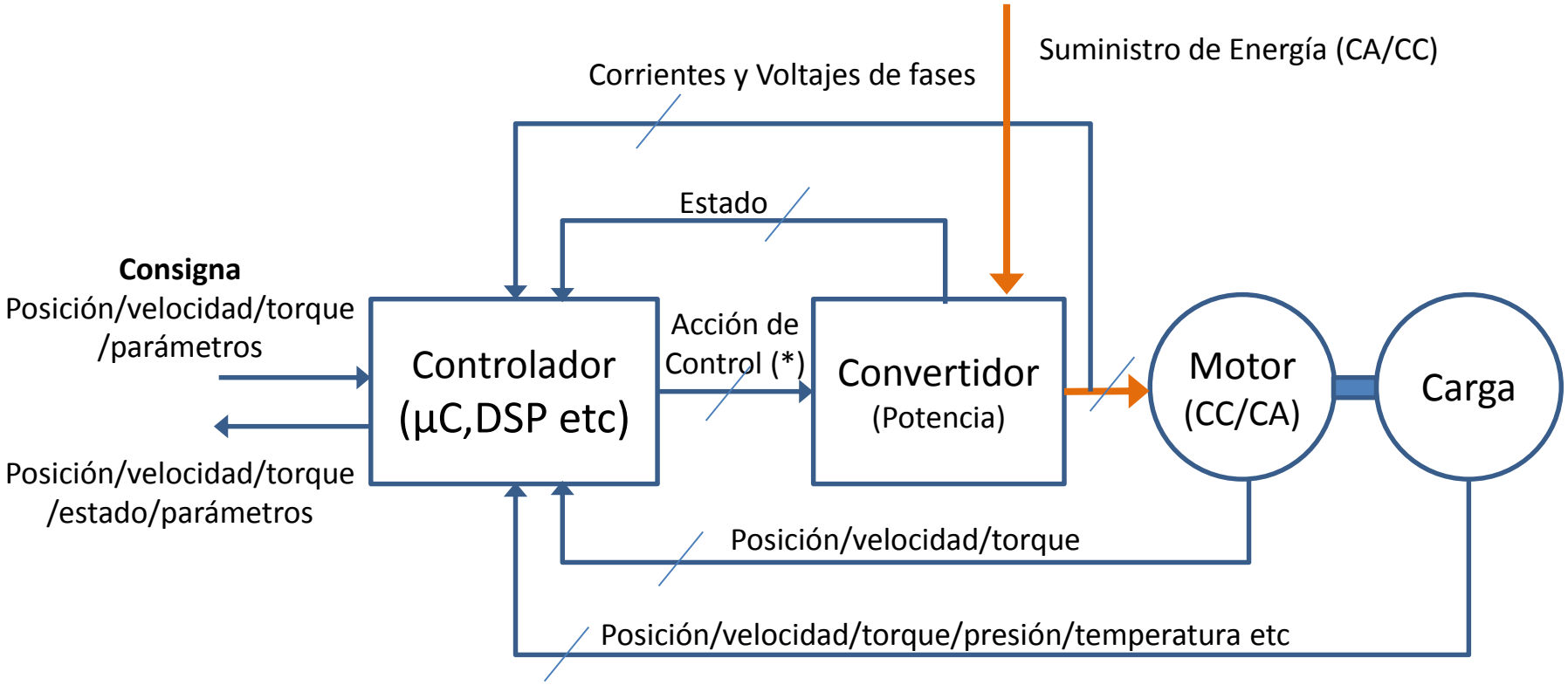
Salidas:

PWM

***Power* PWM**

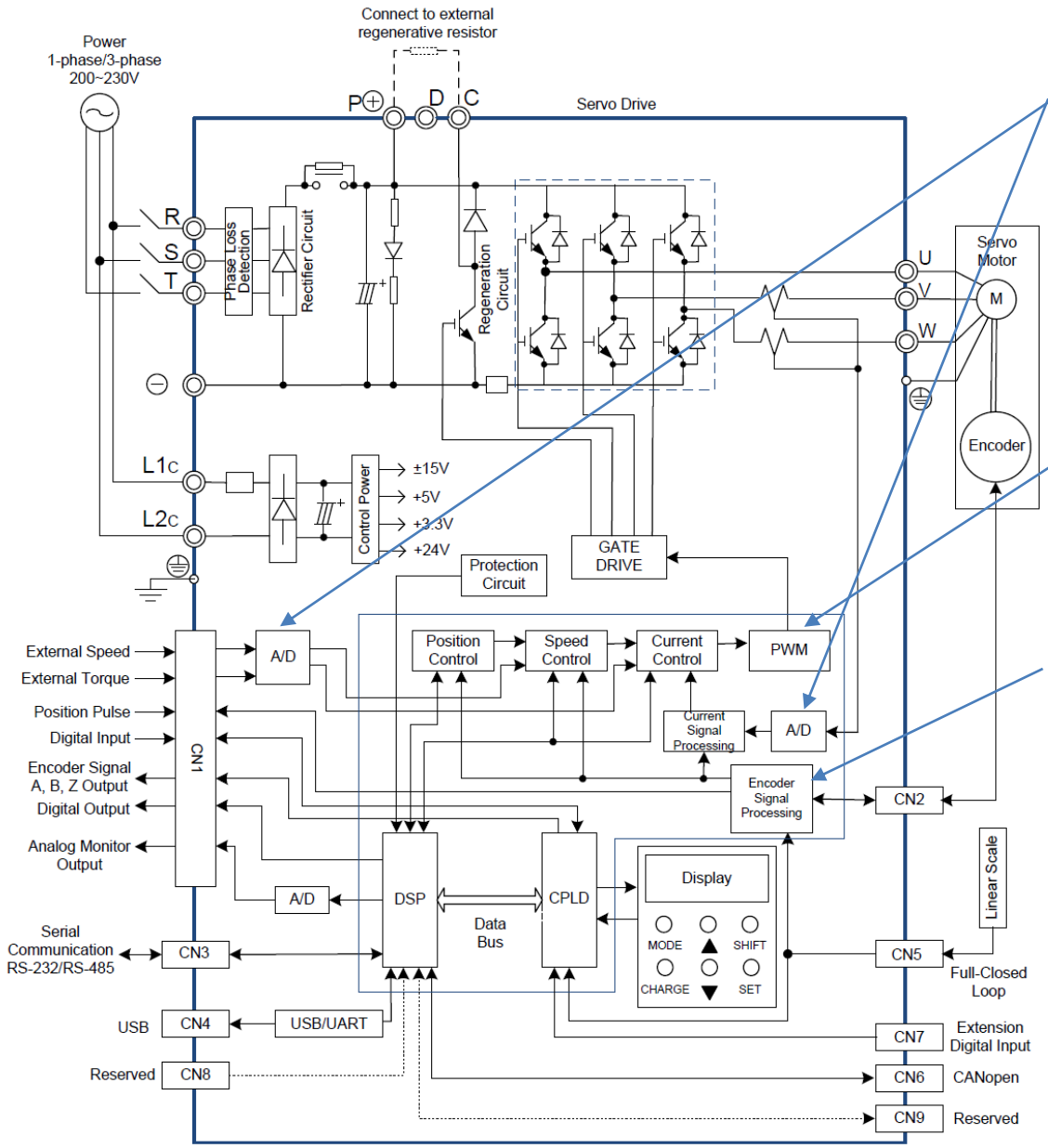
Permiten comandar las distintas topologías de transistores para controlar posición/velocidad/torque de un motor BDC, BLDC, AC asíncrono, *Stepper* (PaP) etc.

Subsistemas para Control de Movimiento



(*) La acción de control comprende la conmutación (encendido/apagado) de los dispositivos de potencia (Tiristores, MOSFETs, IGBTs etc) del convertidor

Esquema del controlador DELTA ASDA-A2 (Feb 2014)



Las **entradas analógicas** del µC miden corriente (para determinar torque), voltaje (para detectar fuerza contra-electromotriz o *Back EMF*) y consignas de Velocidad y/o Torque dadas en forma de tensión.

Las salidas del µC, directas o moduladas (**PWM**) conmutan, a través de "*gate drivers*", los transistores que comandan el motor.

Las entradas de **QEI**, **Timers** conectados en modo contador o medición precisa de tiempos, y **entradas de captura** para sensores Hall permiten determinar directa o indirectamente posición y velocidad del eje, para realizar su control

Los algoritmos de control dependen del tipo de motor y de la aplicación (control en velocidad, en posición y/o en torque). Según el tipo de control deben responder en

CCP

CCP

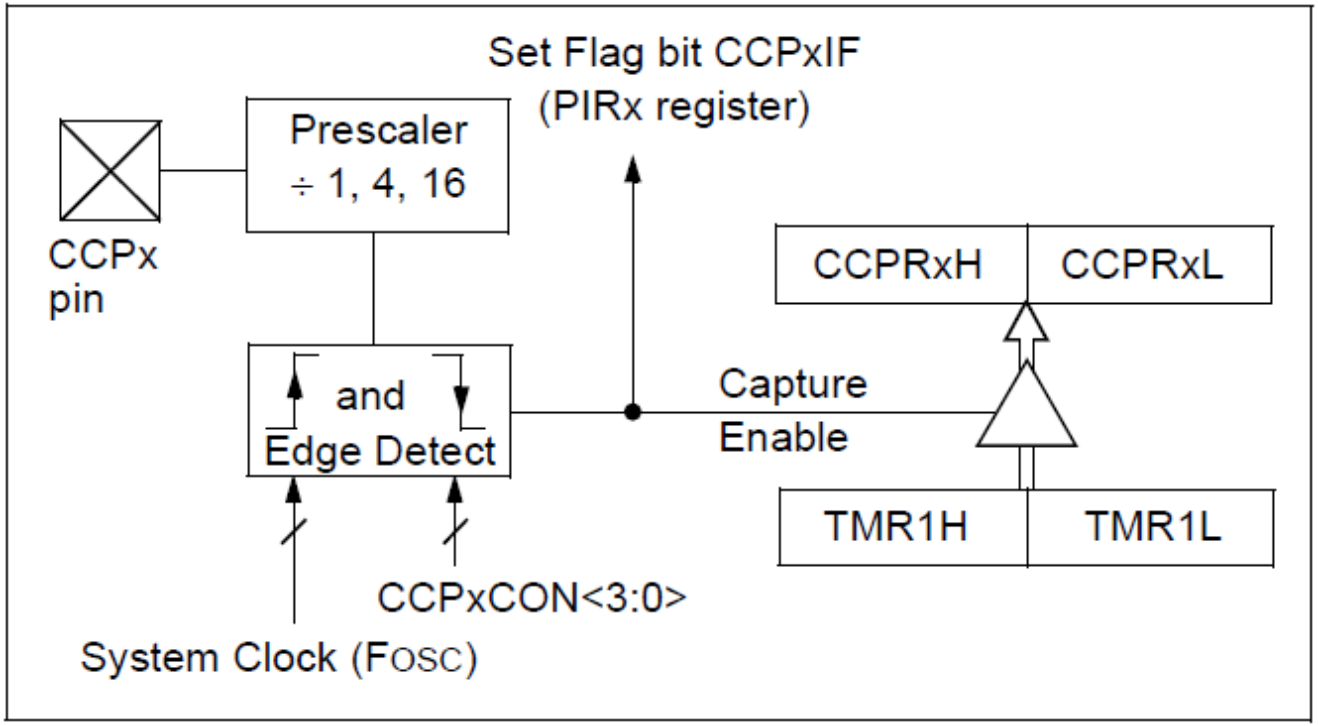
CCP (Capture/Compare/PWM) es un subsistema para generar y medir eventos con precisión temporal.

Modo Capture: Permite medir períodos o anchos de pulsos con precisión.
Se captura en el registro CCPR1 (par CCPR1H:CCPR1L) el valor del Timer1 cuando ocurre alguno de los siguientes eventos en el pin CCP1: flanco ascendente, flanco descendente, 4 flancos ascendentes, 16 flancos ascendentes.

Modo Compare: Permite generar pulsos o eventos internos de duración precisa.
Puede activar/desactivar el pin CCP1, generar una interrupción, resetear Timer1 o iniciar una conversión A/D, cuando el Timer1 iguala el valor del registro CCPR1.

Modo PWM: Permite generar una señal PWM.
La señal PWM se genera en el pin CCP1. Utiliza Timer2 como Base de Tiempo.
El período del PWM se define con el registro PR2 (Timer2 se compara con PR2 y se resetea al igualarse), el y duty cycle está dado por el registro CCPR1 (ó CCPR2)

CCP: Modo Capture

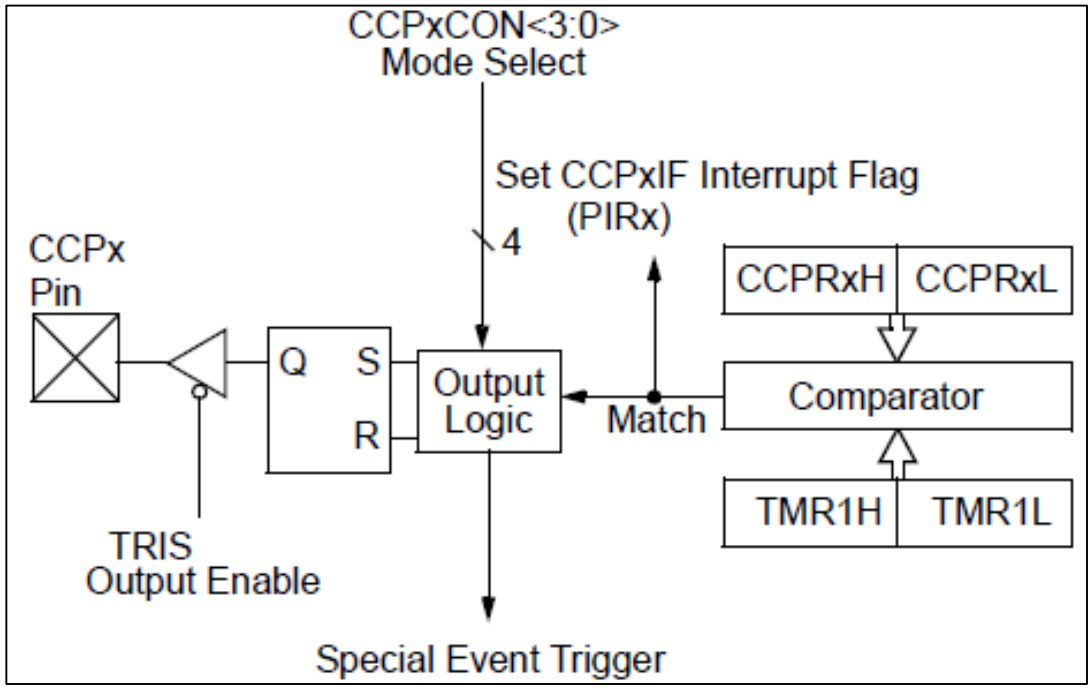


Modo Capture: Permite medir períodos o anchos de pulsos con precisión.
Se captura en el par de registros CCPR1H:CCPR1L el valor TMR1H:TMR1L.

El evento capturado se elige en CCP1CON<3:0> (campo CCP1CON.CCP1M)

0100 = Modo Captura, cada flanco de bajada
0101 = Modo Captura, cada flanco de subida
0110 = Modo Captura, cada 4to flanco de subida
0111 = Modo Captura, cada 16to flanco de subida

CCP: Modo Compare

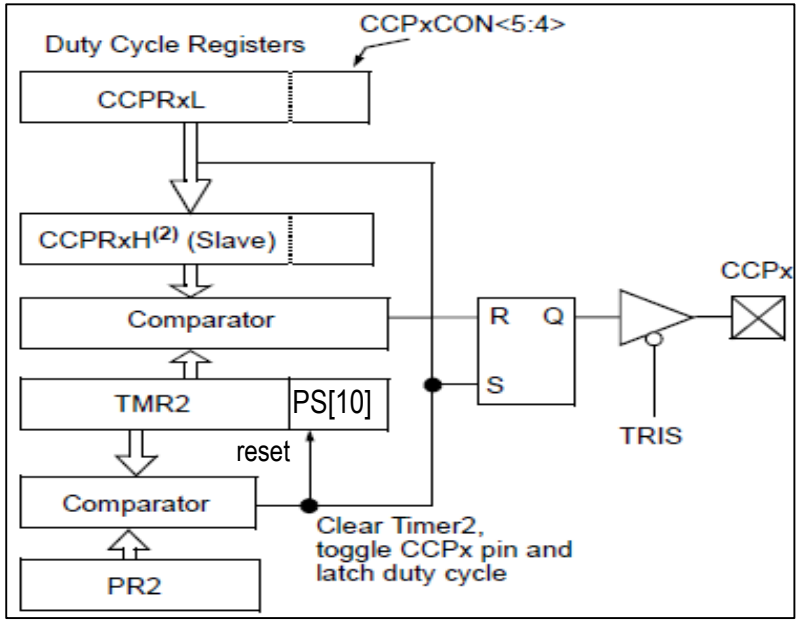


Modo Compare: Permite generar pulsos o eventos internos de duración precisa. El evento se produce cuando el Timer1 iguala el valor del registro CCPR1.

El evento producido se elige en CCP1CON<3:0> (campo CCP1CON.CCP1M)

0010 = Invierte pin CCP1 (y activa flag CCP1IF)
1000 = Activa pin CCP1 (y activa flag CCP1IF)
1001 = Desactiva pin CCP1 (y activa flag CCP1IF)
1010 = Activa CCP1IF, no afecta pin CCP1.
1011 = Activa CCP1IF y resetea Timer1 (no dispara flag TMR1IF)

CCP: Modo PWM



Modo PWM: Permite generar una señal PWM.
La señal PWM se genera en el pin CCP1.
Utiliza Timer2 como Base de Tiempo.
El período del PWM se define con el registro PR2 (Timer2 se compara con PR2 y se resetea al igualarse), el duty cycle está dado por el registro CCPR1 (ó CCPR2)

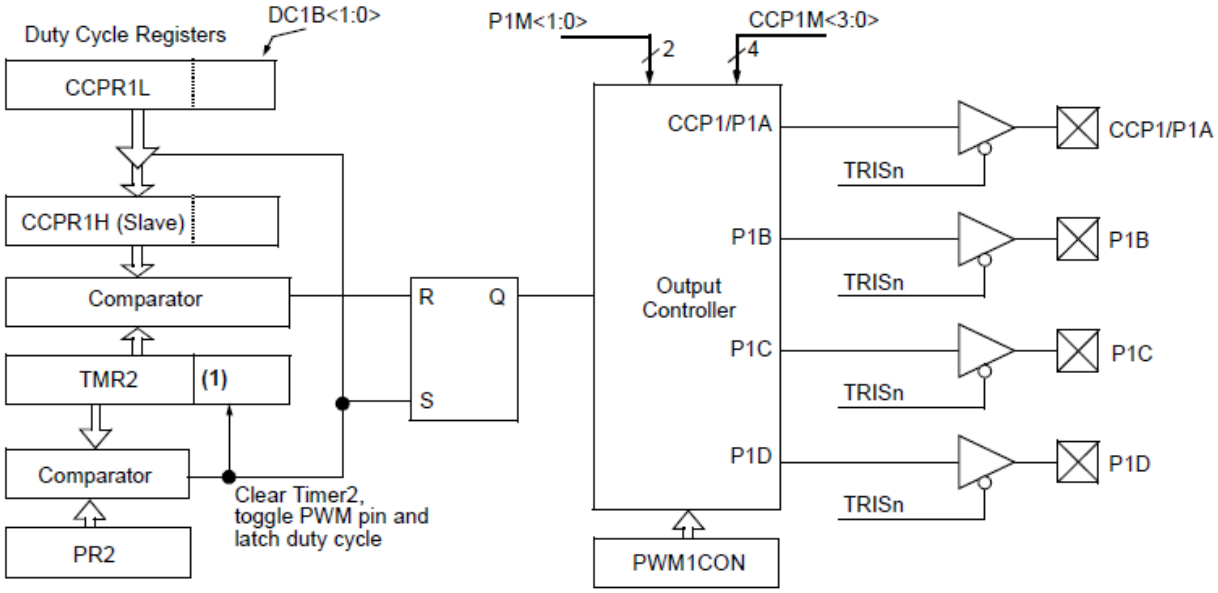
TMR2 es de 8 bits, pero se concatena con 2 bits de prescaler (que llamamos PS[1:0]) para formar una base de tiempo de 10 bits.
PR2 determina el período de TMR2
El programa puede escribir el CCPR1L (8 bits) más los bits 5-4 de CCP1CON para dar el valor de *Duty Cycle.*
CCPR1H es esclavo (se carga con el valor de CCPR1L al finalizar la rampa anterior.
La salida CCP1 se activa cuando TMR2(9:2)=PR2 y se desactiva cuando TMR(9:2)+PS[1:0] = CCPR1H

El módulo CCP2 usa el mismo TMR2, pero los registros CCPR2L/CCPR2H y CCP2CON) y la salida CCP2

Periodo T_{PWM} = [(*PR2*) + *1*] . *4* . T_{OSC} . *(TMR2 Presc)*
Ejemplo: Fosc=10MHz, T_{OSC} *=0.1 us, TMR2 Presc=1, PR2=249* → T_{PWM}*=100 us (*f_{PWM}*=10kHz)*
En este ejemplo el Duty Cycle puede ir de 0 a 999.

CCP: Modo PWM

En el 16F884 y otros similares contemplan la posibilidad de que el PWM se aplique a un puente monofásico de transistores (puente de 4 transistores). Este modo se denomina *Enhanced* PWM (PWM aumentado).



En el 16F88x las salidas P1A (CCP1), P1B, P1C y P1D se pueden configurar en 4 modos de acuerdo con el campo campo CCP1CON.CCP1M:

1100 = PWM simple, modula P1A (salida CCP1)
1101 = PWM medio puente P1A, en fase, P1B en contrafase
1110 = PWM puente completo directo, P
1111 = PWM puente completo P1A, P1C, P1B, P1D en contrafase

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1
—	—	—	STRSYNC	STRD	STRC	STRB	STRA
bit 7							bit 0

CCP: Modo PWM (2)

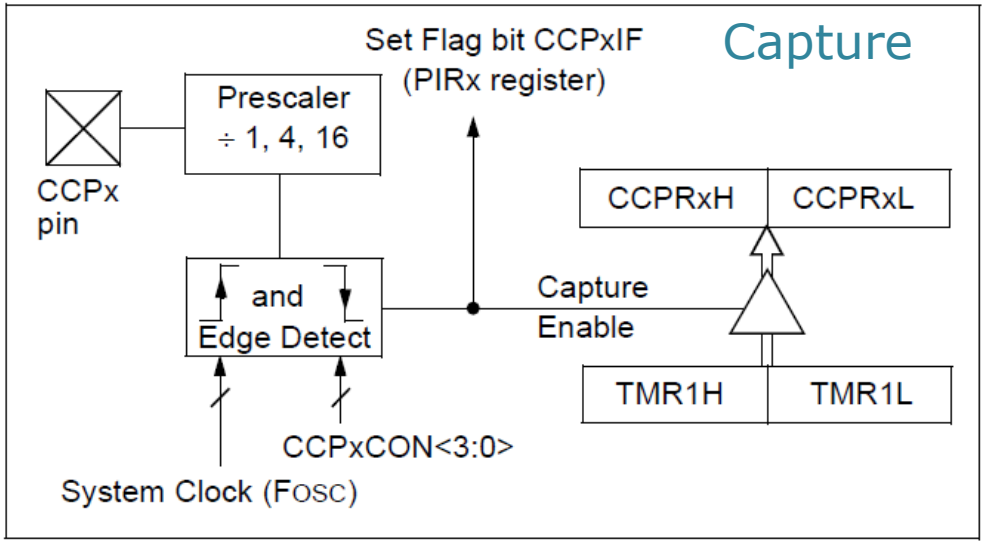
TABLE 11-3: EXAMPLE PWM FREQUENCIES AND RESOLUTIONS (Fosc = 20 MHz)

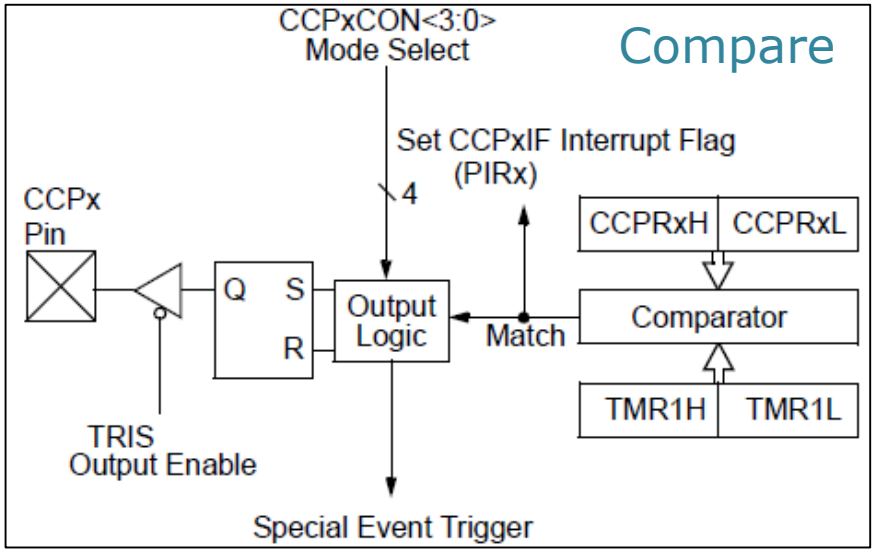
PWM Frequency	1.22 kHz	4.88 kHz	19.53 kHz	78.12 kHz	156.3 kHz	208.3 kHz
Timer Prescale (1, 4, 16)	16	4	1	1	1	1
PR2 Value	0xFF	0xFF	0xFF	0x3F	0x1F	0x17
Maximum Resolution (bits)	10	10	10	8	7	6.6

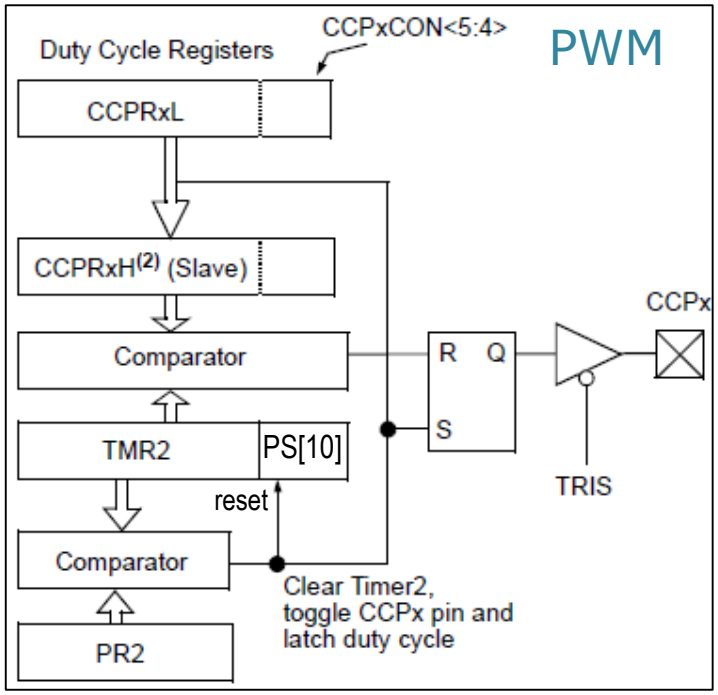
TABLE 11-4: EXAMPLE PWM FREQUENCIES AND RESOLUTIONS (Fosc = 8 MHz)

PWM Frequency	1.22 kHz	4.90 kHz	19.61 kHz	76.92 kHz	153.85 kHz	200.0 kHz
Timer Prescale (1, 4, 16)	16	4	1	1	1	1
PR2 Value	0x65	0x65	0x65	0x19	0x0C	0x09
Maximum Resolution (bits)	8	8	8	6	5	5

CCP:Resumen







CCP1M<3:0>

0 = Capture/Compare/PWM off (resetea módulo)
1 = Reservado
2 = Compara, invierte CCP1 (CCP1IF=1)
3 = Reservado
4 = Captura, flanco bajada
5 = Captura, flanco subida
6 = Captura, 4to flanco subida
7 = Captura, 16to flanco subida
8 = Compara, activa CCP1 (CCP1IF='1')
9 = Compara, desactiva CCP1 (CCP1IF='1')
10 = Compara, CCP1IF=1 (bit CCP1 no cambia)
11 = Compare, dispara evento especial (CCP1IF=1, TMR1=0)
12 = PWM; P1A, P1C activo-high; P1B, P1D activo-high
13 = PWM; P1A, P1C activo-high; P1B, P1D activo-low
14 = PWM; P1A, P1C activo-low; P1B, P1D activo-high
15 = PWM; P1A, P1C activo-low; P1B, P1D activo-low

CCP

REGISTER 11-1: CCP1CON: ENHANCED CCP1 CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
bit 7							bit 0

REGISTER 11-2: CCP2CON: CCP2 CONTROL REGISTER

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0
bit 7							bit 0

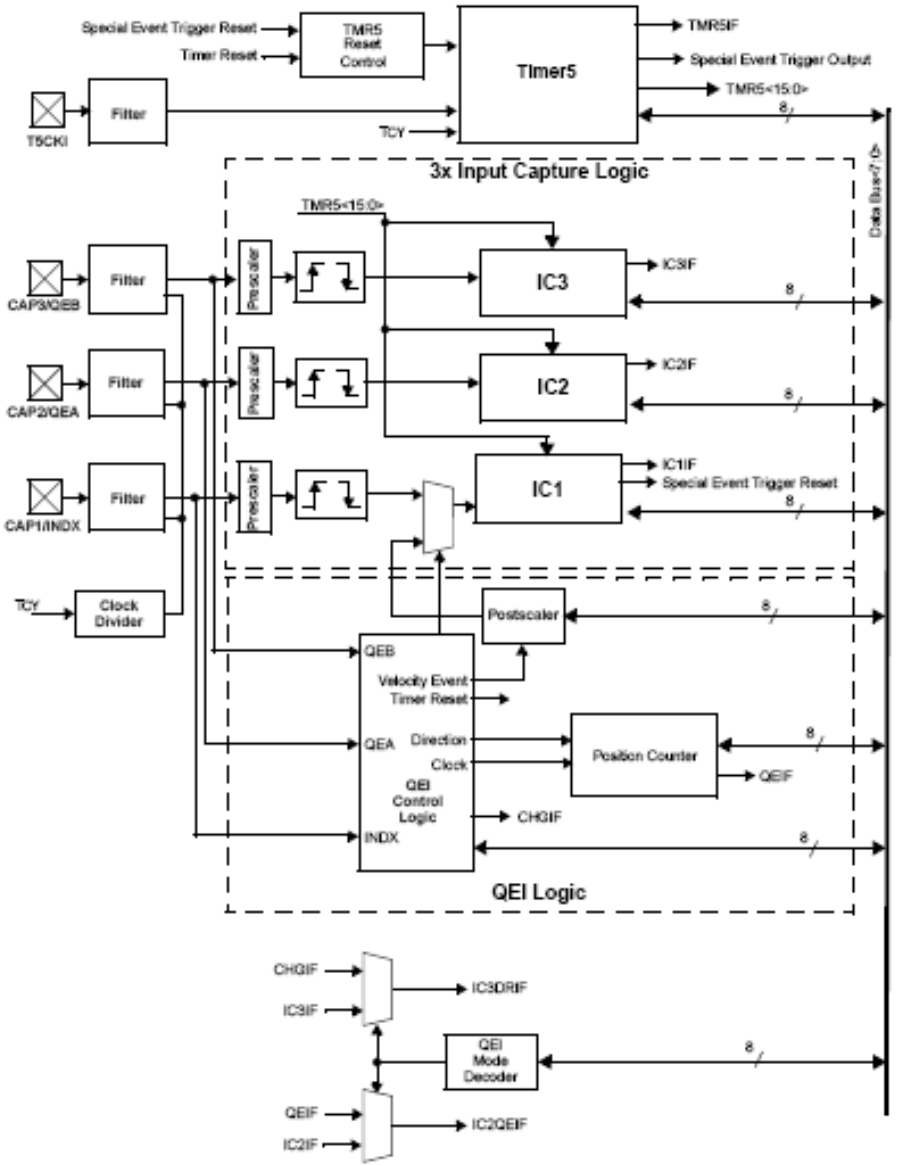
DC1B<1:0>: PWM Duty Cycle Least Significant bits

00 = Single output; P1A modulated; P1B, P1C, P1D assigned as port pins
01 = Full-Bridge output forward; P1D modulated; P1A active; P1B, P1C inactive
10 = Half-Bridge output; P1A, P1B modulated with dead-band control; P1C, P1D assigned as port pins
11 = Full-Bridge output reverse; P1B modulated; P1C active; P1A, P1D inactive

QEI / IC

Módulos de realimentación de movimiento

PIC18F4431



Módulos de realimentación de movimiento

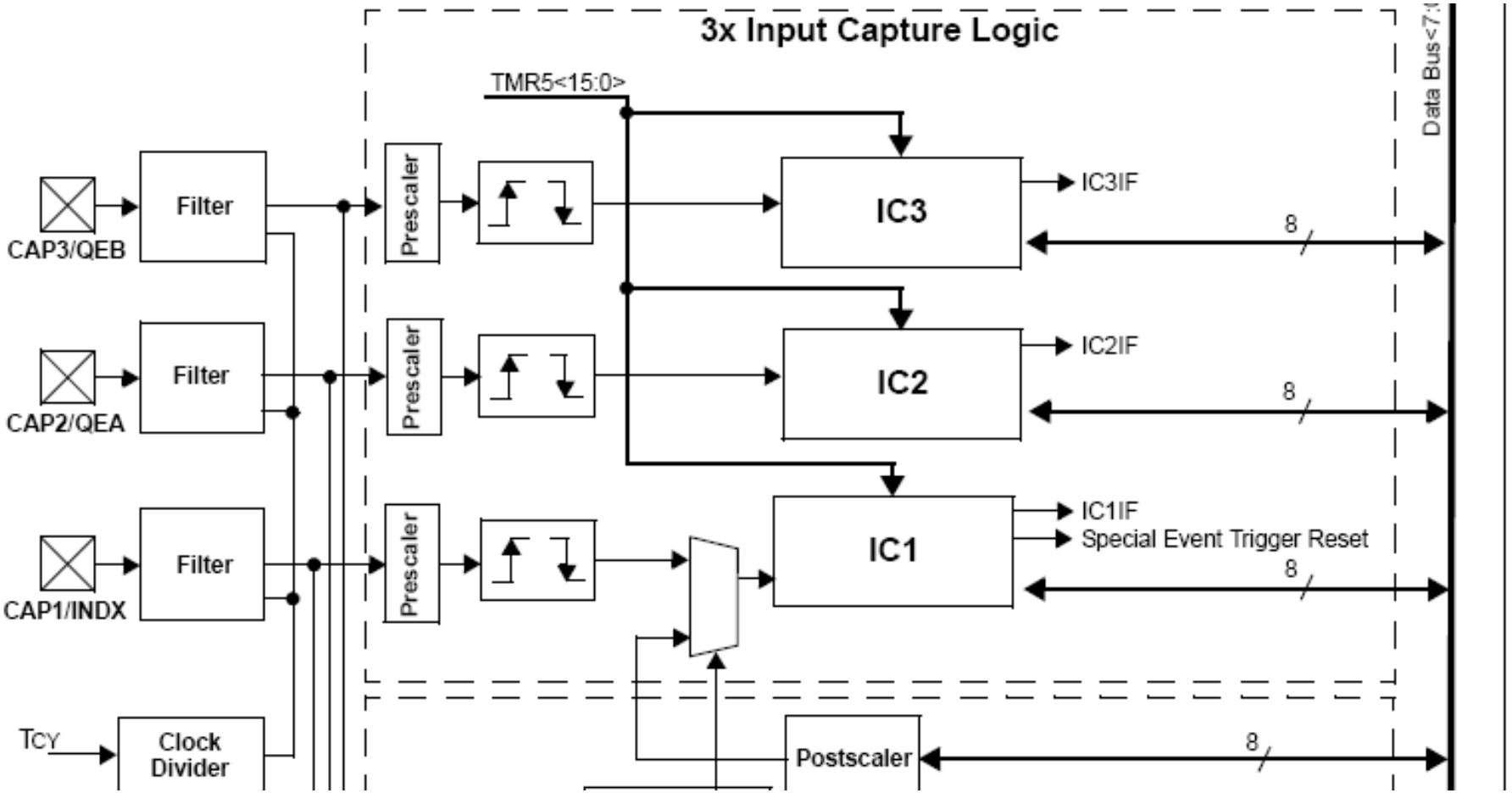
TABLE 17-1: SUMMARY OF MOTION FEEDBACK MODULE FEATURES

Submodule	Mode(s)	Features	Timer	Function
IC (3x)	• Synchronous • Input Capture	• Flexible Input Capture modes • Available Prescaler • Selectable Time Base Reset • Special Event Trigger for ADC Sampling/Conversion or Optional TMR5 Reset Feature (CAP1 only) • Wake-up from Sleep function • Selectable Interrupt Frequency • Optional Noise Filter	TMR5	• 3x Input Capture (edge capture, pulse width, period measurement, capture on change) • Special Event Triggers the A/D Conversion on the CAP1 Input
QEI	QEI	• Detect Position • Detect Direction of Rotation • Large Bandwidth (FCY/16) • Optional Noise Filter	16-Bit Position Counter	• Position Measurement • Direction of Rotation Status
	Velocity Measurement	• 2x and 4x Update modes • Velocity Event Postscaler • Counter Overflow Flag for Low Rotation Speed • Utilizes Input Capture 1 Logic (IC1) • High and Low Velocity Support	TMR5	• Precise Velocity Measurement • Direction of Rotation Status

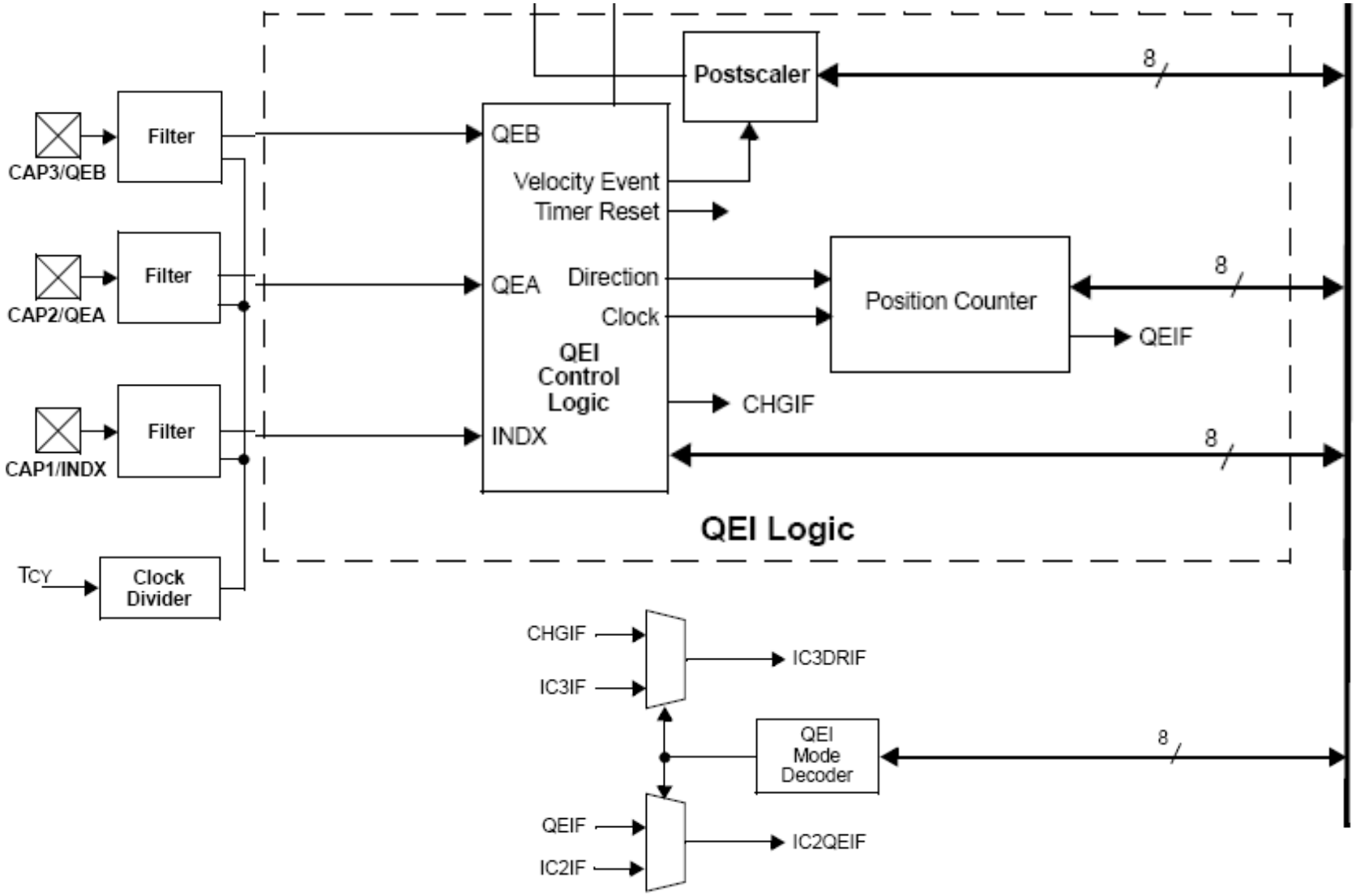
Registros asociados

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	54
IPR3	—	—	—	PTIP	IC3DRIP	IC2QEIP	IC1IP	TMR5IP	56
PIE3	—	—	—	PTIE	IC3DRIE	IC2QEIE	IC1IE	TMR5IE	56
PIR3	—	—	—	PTIF	IC3DRIF	IC2QEIF	IC1IF	TMR5IF	56
TMR5H	Timer5 Register High Byte								57
TMR5L	Timer5 Register Low Byte								57
PR5H	Timer5 Period Register High Byte								57
PR5L	Timer5 Period Register Low Byte								57
T5CON	T5SEN	$\overline{\text{RESEN}}$	T5MOD	T5PS1	T5PS0	$\overline{\text{T5SYNC}}$	TMR5CS	TMR5ON	57
CAP1BUFH/ VELRH	Capture 1 Register High Byte/Velocity Register High Byte(1)								58
CAP1BUFL/ VELRL	Capture 1 Register Low Byte/Velocity Register Low Byte(1)								58
CAP2BUFH/ POSCNTH	Capture 2 Register High Byte/QEI Position Counter Register High Byte(1)								58
CAP2BUFL/ POSCNTL	Capture 2 Register Low Byte/QEI Position Counter Register Low Byte(1)								58
CAP3BUFH/ MAXCNTH	Capture 3 Register High Byte/QEI Max. Count Limit Register High Byte(1)								58
CAP3BUFL/ MAXCNTL	Capture 3 Register Low Byte/QEI Max. Count Limit Register Low Byte(1)								58
CAP1CON	—	CAP1REN	—	—	CAP1M3	CAP1M2	CAP1M1	CAP1M0	59
CAP2CON	—	CAP2REN	—	—	CAP2M3	CAP2M2	CAP2M1	CAP2M0	59
CAP3CON	—	CAP3REN	—	—	CAP3M3	CAP3M2	CAP3M1	CAP3M0	59
DFLTCON	—	FLT4EN	FLT3EN	FLT2EN	FLT1EN	FLTCK2	FLTCK1	FLTCK0	59
QEICON	$\overline{\text{VELM}}$	QERR	UP/$\overline{\text{DOWN}}$	QEIM2	QEIM1	QEIM0	PDEC1	PDEC0	56

Lógica de captura (input capture)

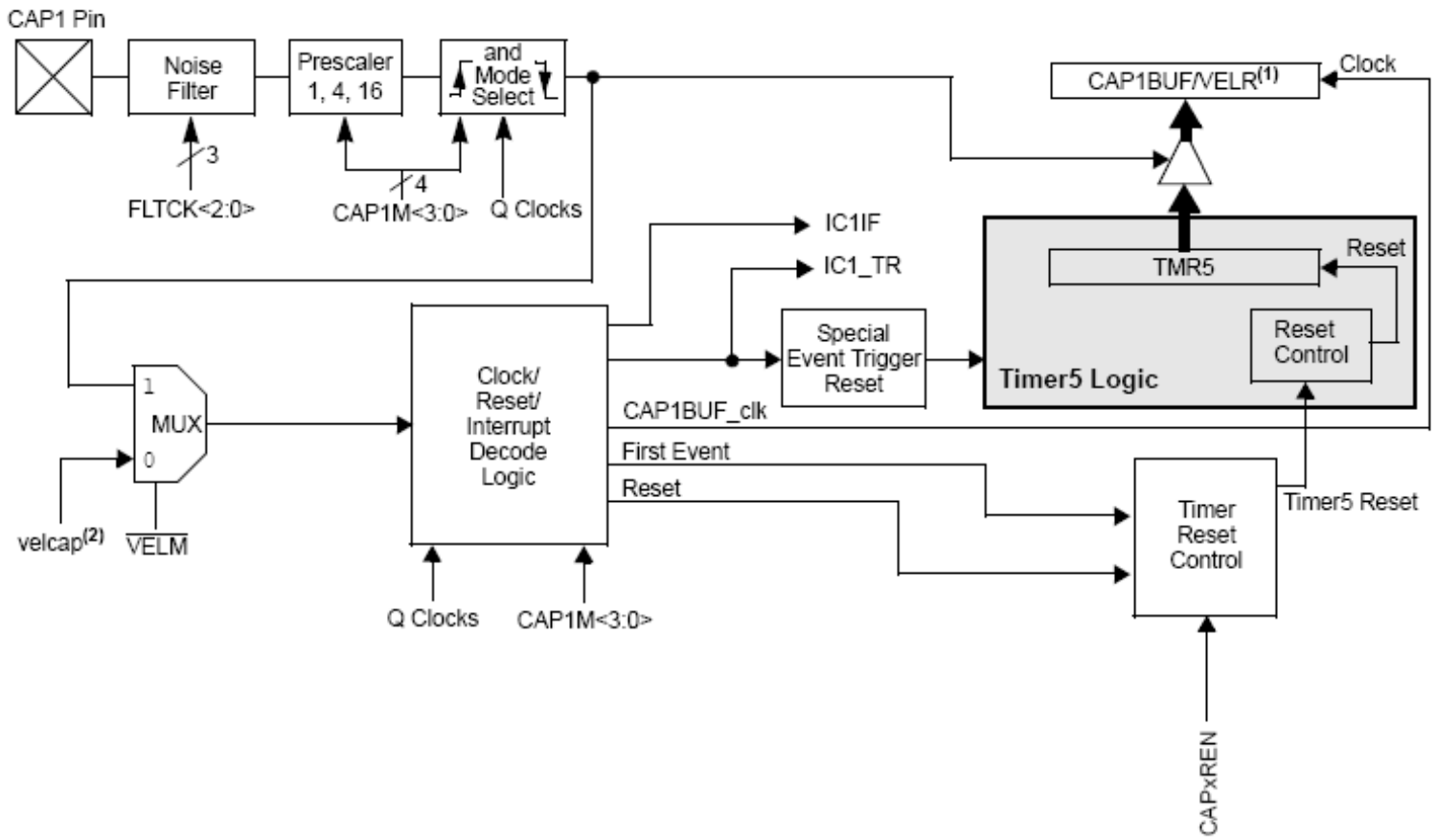


Lógica QEI (interfaz de encoder)



IC1

FIGURE 17-2: INPUT CAPTURE BLOCK DIAGRAM FOR IC1

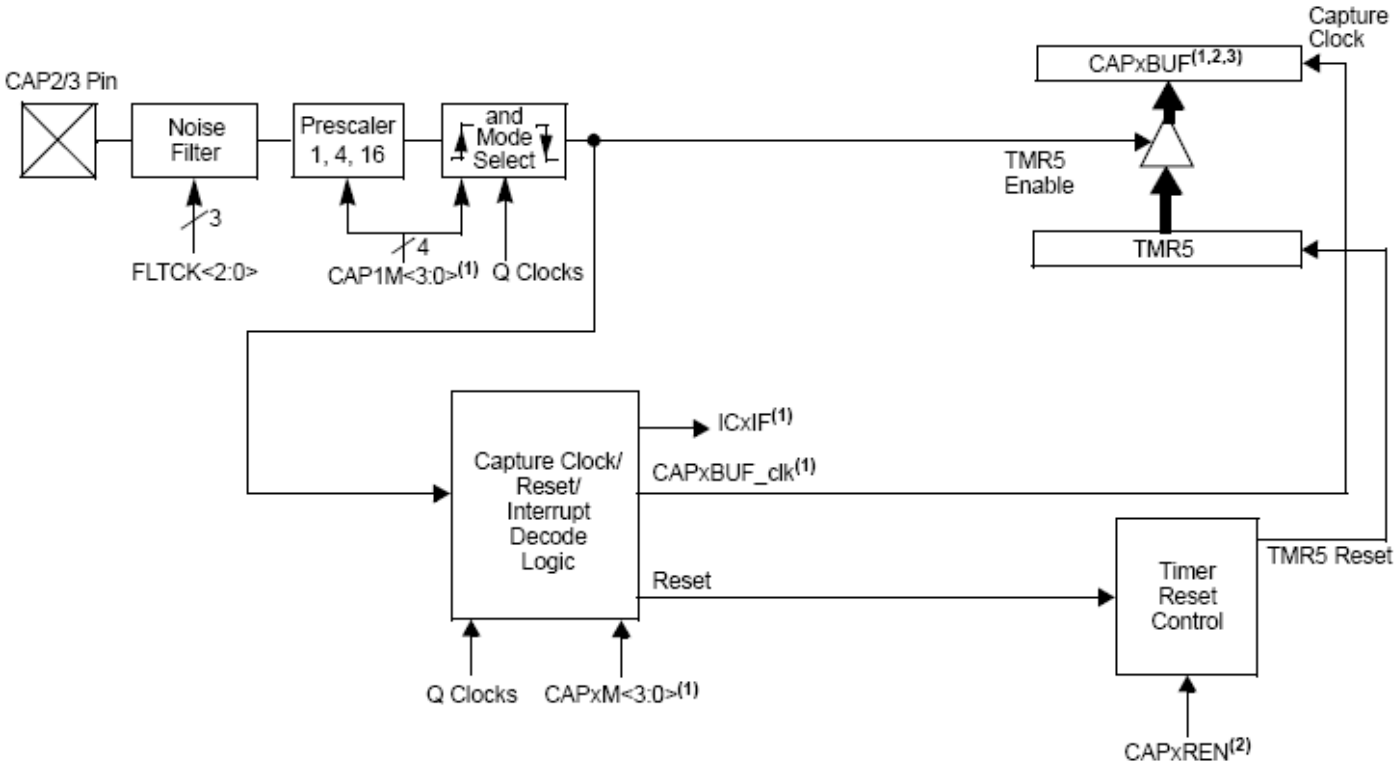


Note 1: CAP1BUF register is reconfigured as VELR register when QEI mode is active.

2: QEI generated velocity pulses, vel_out, are downsampled to produce this velocity capture signal.

IC2 e IC3

FIGURE 17-3: INPUT CAPTURE BLOCK DIAGRAM FOR IC2 AND IC3



Note 1: IC2 and IC3 are denoted as x = 2 and 3.

2: CAP2BUF is enabled as POSCNT when QEI mode is active.

3: CAP3BUF is enabled as MAXCNT when QEI mode is active.

CAP1CON, CAP2CON, CAP3CON

REGISTER 17-1: CAPxCON: INPUT CAPTURE x CONTROL REGISTER

U-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
—	CAPxREN	—	—	CAPxM3	CAPxM2	CAPxM1	CAPxM0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **Unimplemented:** Read as '0'

bit 6 **CAPxREN:** Time Base Reset Enable bit
- 1 = Enabled
- 0 = Disable selected time base Reset on capture

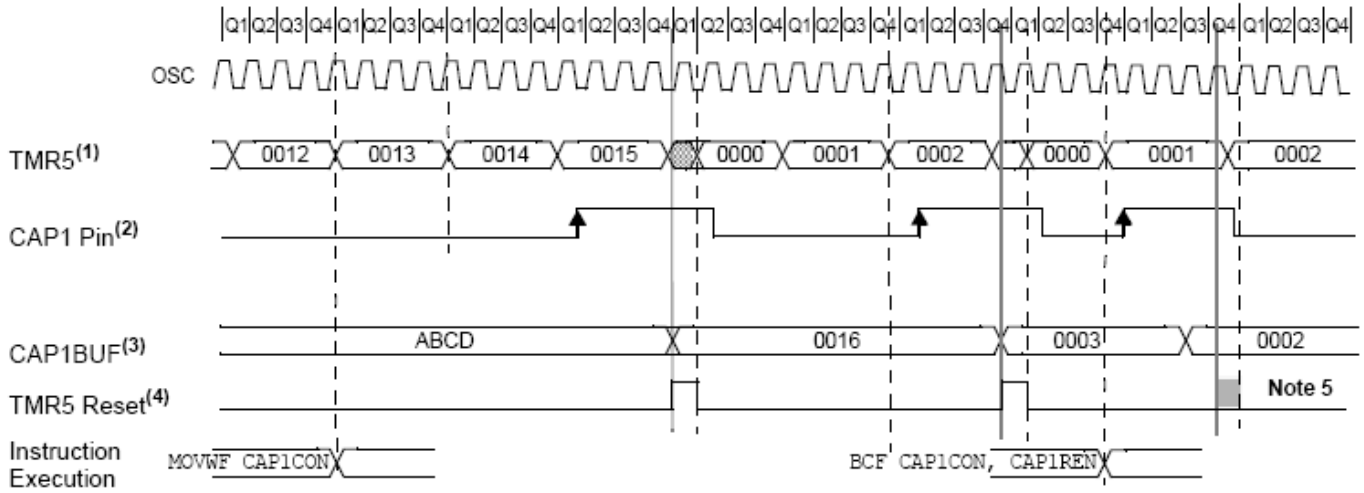
bit 5-4 **Unimplemented:** Read as '0'

bit 3-0 **CAPxM<3:0>:** Input Capture x (ICx) Mode Select bits
- 1111 = Special Event Trigger mode; the trigger occurs on every rising edge on CAP1 input(1)
- 1110 = Special Event Trigger mode; the trigger occurs on every falling edge on CAP1 input(1)
- 1101 = Unused
- 1100 = Unused
- 1011 = Unused
- 1010 = Unused
- 1001 = Unused
- 1000 = Capture on every CAPx input state change
- 0111 = Pulse-Width Measurement mode, every rising to falling edge
- 0110 = Pulse-Width Measurement mode, every falling to rising edge
- 0101 = Frequency Measurement mode, every rising edge
- 0100 = Capture mode, every 16th rising edge
- 0011 = Capture mode, every 4th rising edge
- 0010 = Capture mode, every rising edge
- 0001 = Capture mode, every falling edge
- 0000 = Input Capture x (ICx) off

Note 1: Special Event Trigger is only available on CAP1. For CAP2 and CAP3, this configuration is unused.

Modo Captura

FIGURE 17-4: EDGE CAPTURE MODE TIMING



Note 1: TMR5 is a synchronous time base input to the input capture; prescaler = 1:1. It increments on the Q1 rising edge.

2: IC1 is configured in Edge Capture mode (CAP1M<3:0> = 0010) with the time base reset upon edge capture (CAP1REN = 1) and no noise filter.

3: TMR5 value is latched by CAP1BUF on TCY. In the event that a write to TMR5 coincides with an input capture event, the write will always take precedence. All Input Capture Buffers, CAP1BUF, CAP2BUF and CAP3BUF, are updated with the incremented value of the time base on the next TCY clock edge when the capture event takes place (see **Note 4** when Reset occurs).

4: TMR5 Reset is normally an asynchronous Reset signal to TMR5. When used with the input capture, it is active immediately after the time base value is captured.

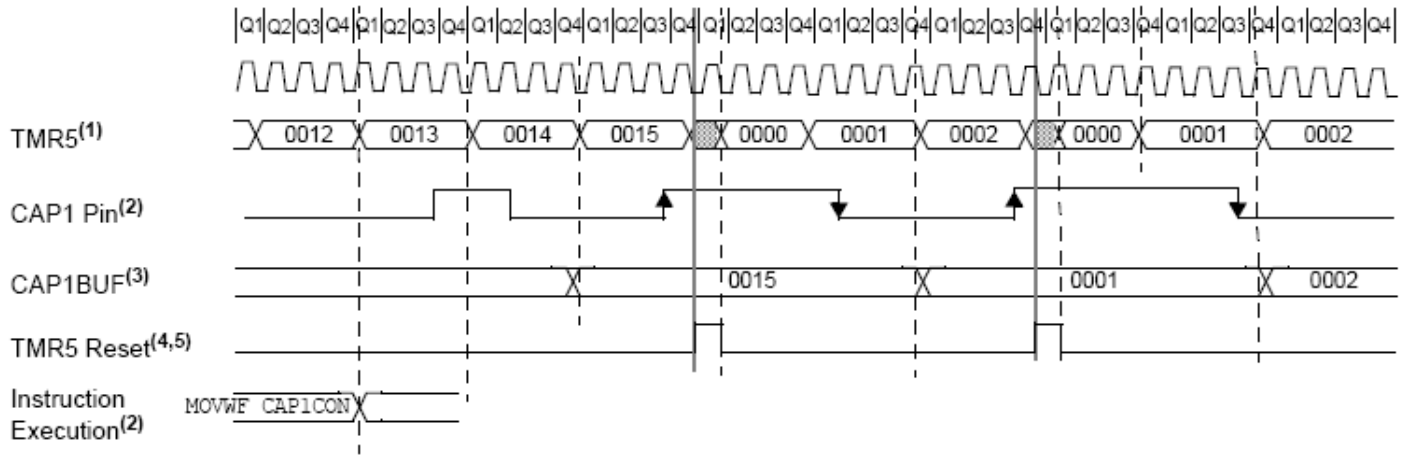
5: TMR5 Reset pulse is disabled by clearing the CAP1REN bit (e.g., BCF CAP1CON, CAP1REN).

En cada flanco de subida, de bajada, 4to de subida, 16to de subida

U-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
—	CAPxREN	—	—	CAPxM3	CAPxM2	CAPxM1	CAPxM0
bit 7							bit 0

Modo Medición de Ancho de Pulso

FIGURE 17-5: PULSE-WIDTH MEASUREMENT MODE TIMING



Note 1: TMR5 is a synchronous time base input to the input capture; prescaler = 1:1. It increments on every Q1 rising edge.

2: IC1 is configured in Pulse-Width Measurement mode (CAP1M<3:0> = 0111, rising to falling pulse-width measurement). No noise filter on CAP1 input is used. The MOVWF instruction loads CAP1CON when W = 0111.

3: TMR5 value is latched by CAP1BUF on TCY rising edge. In the event that a write to TMR5 coincides with an input capture event, the write will always take precedence. All Input Capture Buffers, CAP1BUF, CAP2BUF and CAP3BUF, are updated with the incremented value of the time base on the next TCY clock edge when the capture event takes place (see Note 4 when Reset occurs).

4: TMR5 Reset is normally an asynchronous Reset signal to TMR5. When used in Pulse-Width Measurement mode, it is always present on the edge that first initiates the pulse-width measurement (i.e., when configured in the rising to falling Pulse-Width Measurement mode); it is active on each rising edge detected. In the falling to rising Pulse-Width Measurement mode, it is active on each falling edge detected.

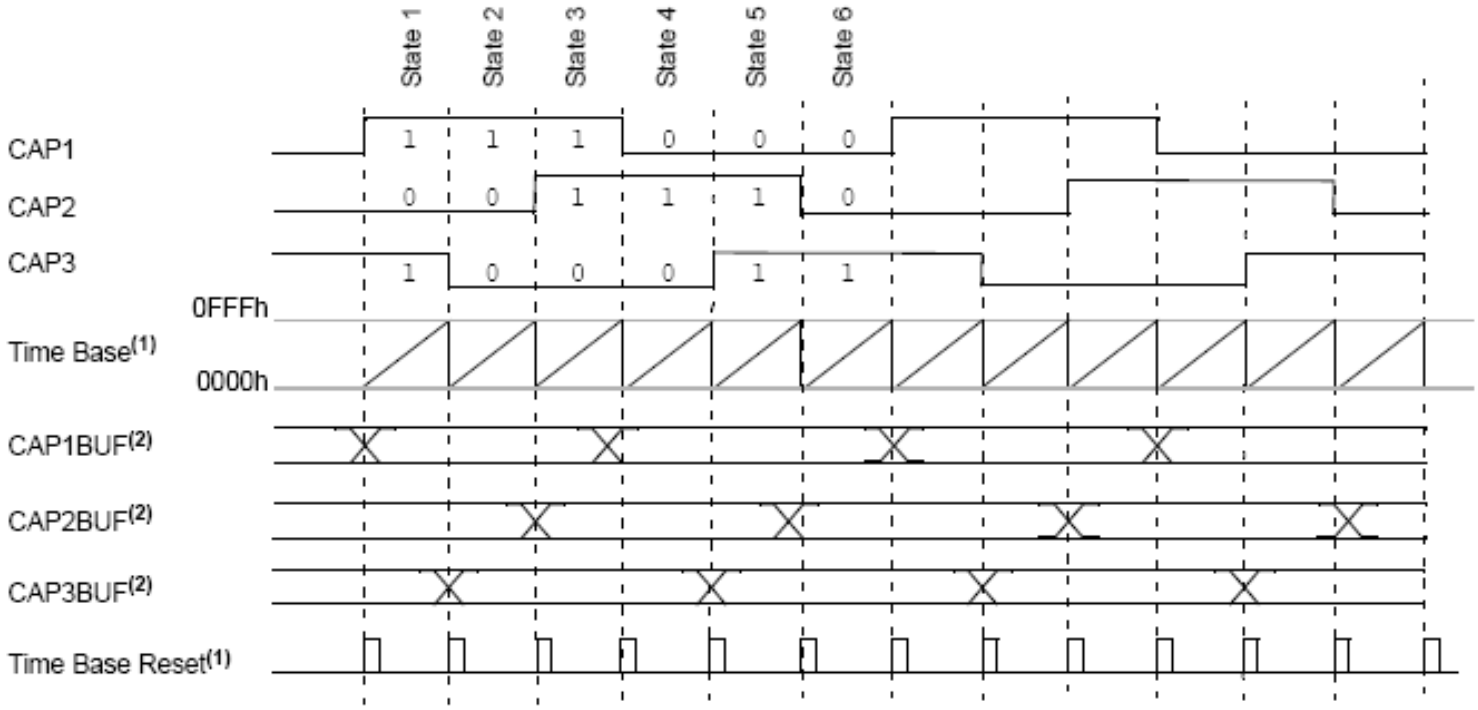
5: TMR5 Reset pulse is activated on the capture edge. The CAP1REN bit has no bearing in this mode.

Mide ancho de pulso alto (flanco de subida a flanco de bajada) **CAPxM<3:0> = 0110)** o bajo (flanco de bajada a flanco de subida) **CAPxM<3:0> = 0111)**

U-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
—	CAPxREN	—	—	CAPxM3	CAPxM2	CAPxM1	CAPxM0
bit 7							bit 0

Modo Captura en cambio de estado (Modo sensor Hall)

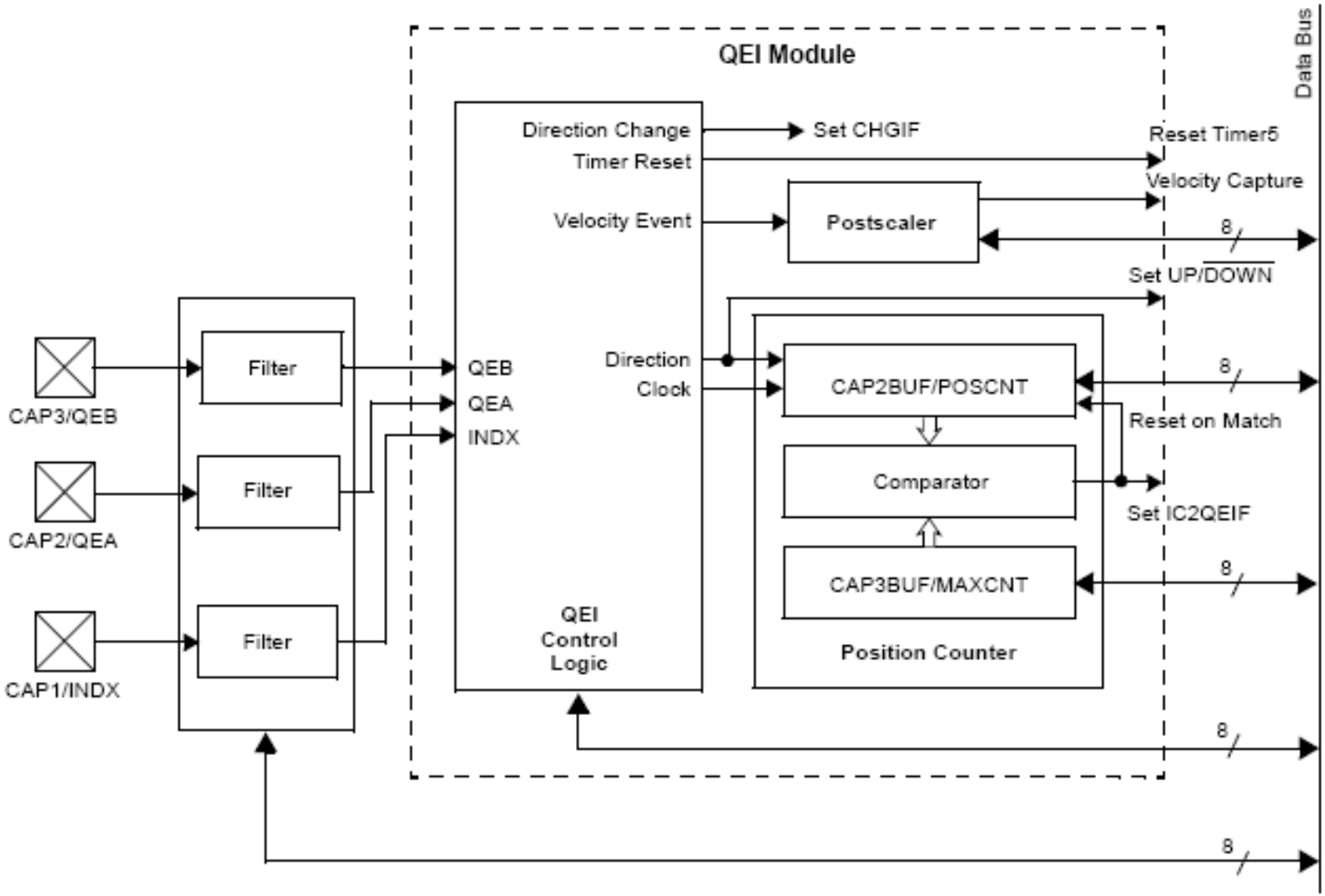
FIGURE 17-6: INPUT CAPTURE ON STATE CHANGE (HALL EFFECT SENSOR MODE)



Note 1: TMR5 can be selected as the time base for input capture. The time base can be optionally reset when the Capture Reset Enable bit is set (CAPxREN = 1).

2: Detailed CAPxBUF event timing (all modes reflect the same capture and Reset timing) is shown in Figure 17-4. There are six commutation BLDC Hall effect sensor states shown. The other two remaining states (i.e., 000h and 111h) are invalid in the normal operation. They remain to be decoded by the CPU firmware in BLDC motor application.

Lógica QEI (detalle)



REGISTER 17-2: QEICON: QUADRATURE ENCODER INTERFACE CONTROL REGISTER

R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
$\overline{\text{VELM}}$	QERR(1)	UP/$\overline{\text{DOWN}}$	QEIM2(2,3)	QEIM1(2,3)	QEIM0(2,3)	PDEC1	PDEC0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **$\overline{\text{VELM}}$:** Velocity Mode bit
- 1 = Velocity mode disabled
- 0 = Velocity mode enabled

bit 6 **QERR:** QEI Error bit(1)
- 1 = Position counter overflow or underflow(4)
- 0 = No overflow or underflow

bit 5 **UP/$\overline{\text{DOWN}}$:** Direction of Rotation Status bit
- 1 = Forward
- 0 = Reverse

bit 4-2 **QEIM<2:0>:** QEI Mode bits(2,3)
- 111 = Unused
- 110 = QEI enabled in 4x Update mode; position counter is reset on period match (POSCNT = MAXCNT)
- 101 = QEI enabled in 4x Update mode; INDX resets the position counter
- 100 = Unused
- 010 = QEI enabled in 2x Update mode; position counter is reset on period match (POSCNT = MAXCNT)
- 001 = QEI enabled in 2x Update mode; INDX resets the position counter
- 000 = QEI off

bit 1-0 **PDEC<1:0>:** Velocity Pulse Reduction Ratio bits
- 11 = 1:64
- 10 = 1:16
- 01 = 1:4
- 00 = 1:1

Note 1: QEI must be enabled and in Index mode.

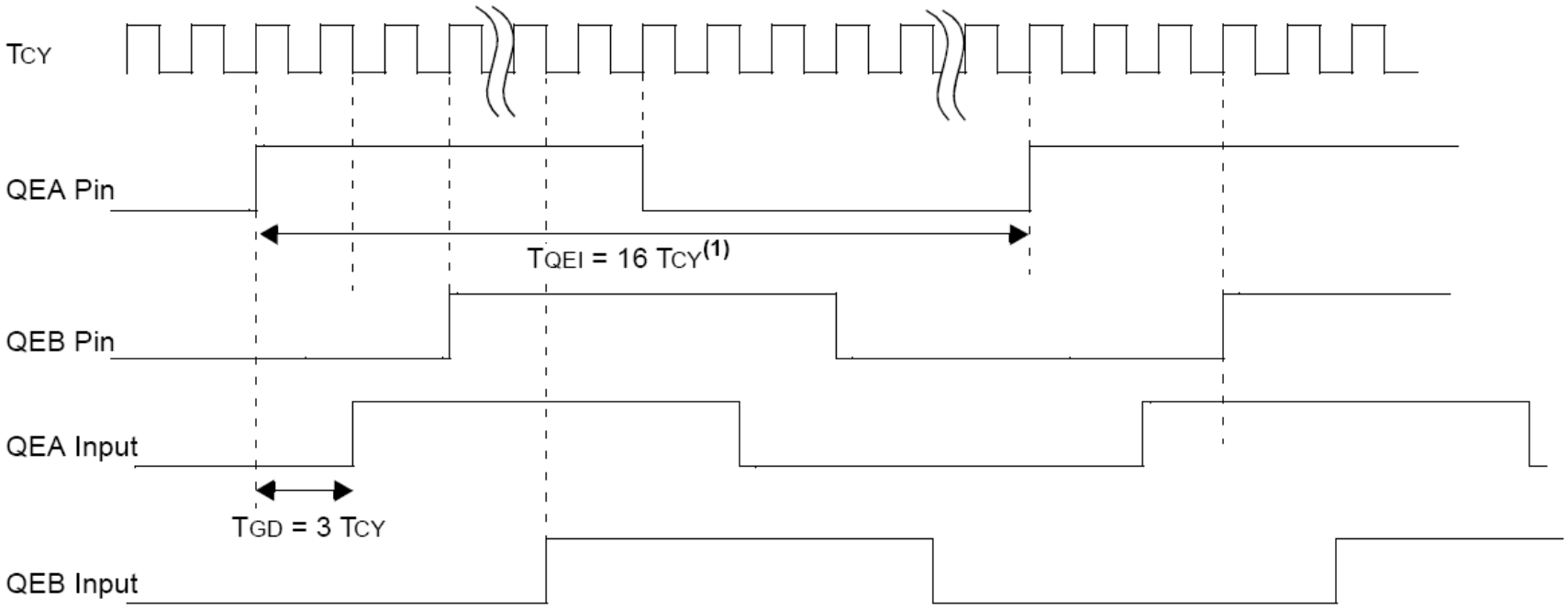
2: QEI mode select must be cleared (= 000) to enable CAP1, CAP2 or CAP3 inputs. If QEI and IC modules are both enabled, QEI will take precedence.

3: Enabling one of the QEI operating modes remaps the IC Buffer registers, CAP1BUFH, CAP1BUFL, CAP2BUFH, CAP2BUFL, CAP3BUFH and CAP3BUFL, as the VELRH, VELRL, POSCNTH, POSCNTL, MAXCNTH and MAXCNTL registers (respectively) for the QEI.

4: The QERR bit must be cleared in software.

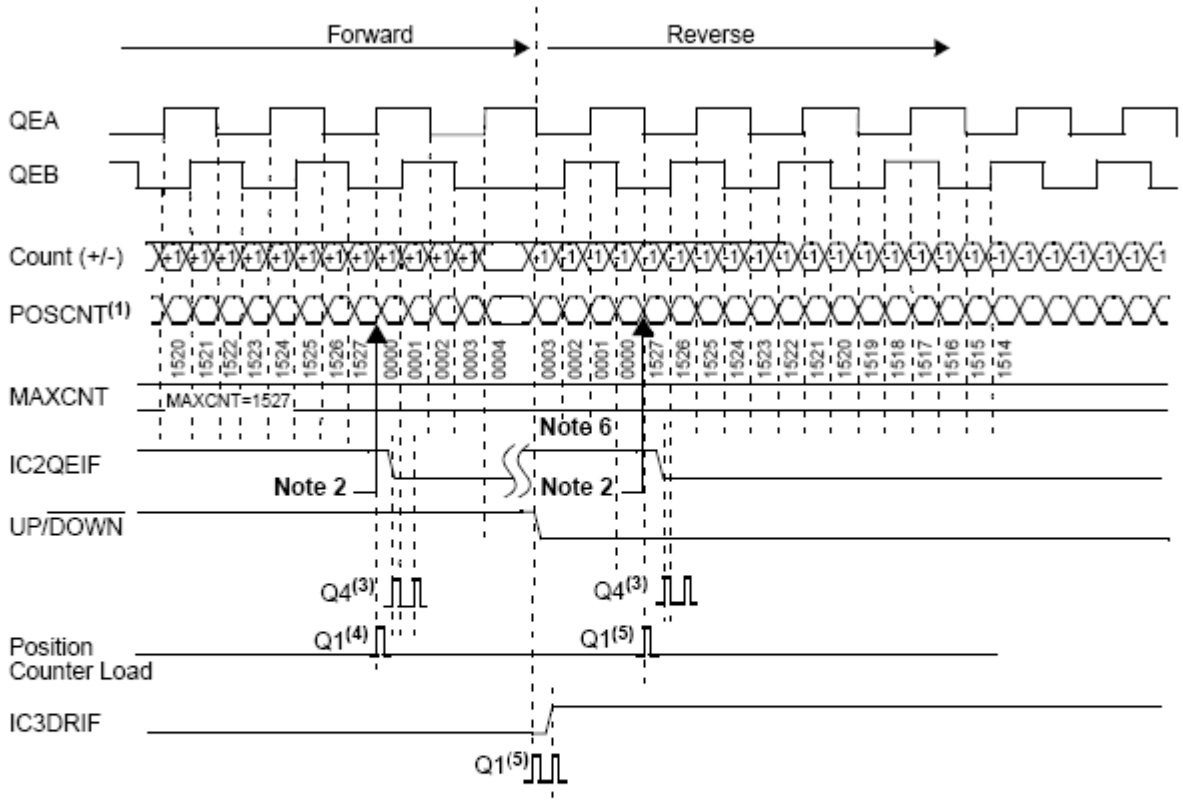
El mínimo período de cada entrada en cuadratura TQEI es 16 TCY siendo TCY=4/FOSC
Por ejemplo, con FOSC 40MHz, TCY=0,1 µs, debe ser TQEI>1,6µs
Esto equivale a 625 kHz para FQEI
Ejercicio: Estimar máxima RPM para un eje con encoders de 1024 líneas y 2048 líneas
La máxima tasa de conteo de posición FPOS podrá ser, en modo 4x:

FPOS = 4.FQEI <FCY/4 → FPOS<FOSC/16



Note 1: The module design allows a quadrature frequency of up to FQEI = FCY/16.

FIGURE 17-10: QEI MODULE RESET TIMING ON PERIOD MATCH



Note 1: The POSCNT register is shown in QEI x4 Update mode (POSCNT increments on every rising and every falling edge of QEA and QEB input signals). Asynchronous external QEA and QEB inputs are synchronized to the TCY clock by the input sampling FF in the noise filter (see Figure 17-14).

2: When POSCNT = MAXCNT, POSCNT is reset to '0' on the next QEA rising edge. POSCNT is set to MAXCNT when POSCNT = 0 (when decrementing), which occurs on the next QEA falling edge.

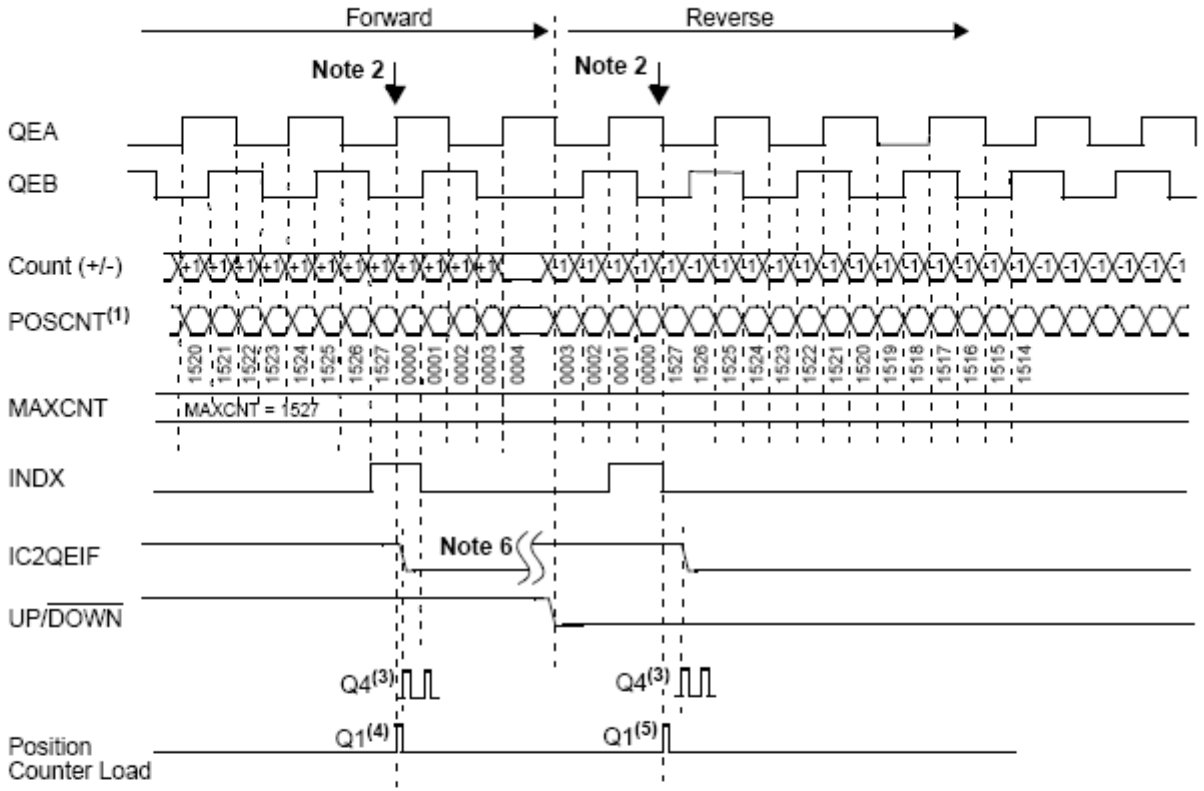
3: IC2QEIF is generated on the Q4 rising edge.

4: Position counter is loaded with '0' (which is a rollover event in this case) on POSCNT = MAXCNT.

5: Position counter is loaded with MAXCNT value (1527h) on underflow.

6: IC2QEIF must be cleared in software.

FIGURE 17-11: QEI MODULE RESET TIMING WITH THE INDEX INPUT



Note 1: POSCNT register is shown in QEI x4 Update mode (POSCNT increments on every rising and every falling edge of QEA and QEB input signals).

2: When an INDX Reset pulse is detected, POSCNT is reset to '0' on the next QEA or QEB edge. POSCNT is set to MAXCNT when POSCNT = 0 (when decrementing), which occurs on the next QEA or QEB edge. a similar Reset sequence occurs for the reverse direction, except that the INDX signal is recognized on its falling edge. The Reset is generated on the next QEA or QEB edge.

3: IC2QEIF is enabled for one TCY clock cycle.

4: The position counter is loaded with 0000h (i.e., Reset) on the next QEA or QEB edge when the INDX is high.

5: The position counter is loaded with a MAXCNT value (e.g., 1527h) on the next QEA or QEB edge following the INDX falling edge input signal detect).

6: IC2QEIF must be cleared in software.

Power PWM

3.D
Control en tiempo real.

Ejecución de múltiples tareas en un procesador

•Un mismo procesador se reparte cíclicamente entre dos o más tareas, con la periodicidad o prioridad que cada tarea demande.
•Cada tarea corre en su "contexto", esto es tiene sus variables y utiliza la CPU como propia.
•La ***conmutación de contexto*** consiste en guardar los datos relevantes cuando se interrumpe la tarea (variables, valores de registros de la CPU) y recuperarlos cuando se retorna a ella.
•Las tareas pueden compartir datos globales, recursos de hardware, y trabajar en forma coordinada para realizar una aplicación más compleja.

- Por ejemplo un procesador de un sistema de control multivariable realiza más de una actividad.
  - Sistema de navegación autónoma basado en mezcla de sensores (odómetros y GPS etc)
  - Robot o supervisor de 2 o más ejes.

Ejecución de múltiples tareas en un procesador

Interrupciones

•Hemos visto una primera técnica de para ejecutar tareas en “paralelo” utilizando **interrupciones.**

•Ejemplo

- •Tarea 1: Recibir consignas por puerto serie (modo, posición, velocidad etc)
- •Tarea 2: Control de movimiento según modo. (homing, ppm…)
- •Tarea 3: Supervisión de modos y estados.

•Las tareas 2 y 3 son periódicas, la tarea 1 se ejecuta con interrupción al recibir nuevo caracter en puerto serie.

•Las interrupciones pueden estar vectorizadas para minimizar las latencias.
•En algunos microcontroladores se puede establecer niveles de prioridad de las interrupciones.

Sistema Operativo

- Partes y funciones de un Sistema Operativo
  - Núcleo (kernel)
    - Multiprogramación: Ejecución de programas o tareas simultáneas.
    - Control del uso de los recursos de hardware (periféricos, memoria) por cada tarea.
  - Interfaz de abstracción del hardware (HAL)
    - Rutinas de servicio para facilitar que las tareas interactúen con el hardware (periféricos)

Tipos de sistema operativo

- Según el rigor temporal:
  - SO de Tiempo Real (RTOS)
  - SO No Tiempo Real
- Según admita o no el replanteo de tareas durante la ejecución.
  - Estático: Se definen las tareas antes de correr.
  - Dinámico: Admite iniciar y concluir tareas durante su ejecución (ej Windows).

- Los RTOS alojados en el *firmware* de sistemas embebidos suelen ser de tipo estático.

Sistema Operativo de Tiempo Real (RTOS)

Constan de un núcleo (kernel) multitarea.
El kernel realiza **la ubicación de las tareas en porciones de tiempo** (*time slices*) apropiadas al tiempo que tarda su ejecución y la periodicidad o prioridad exigidas. En el modo básico puede imaginarse un ciclo en el que se van turnando las distintas tareas, con la necesaria **conmutación de contexto**.
Si una tarea demanda demasiado tiempo de ejecución tal que afecta la periodicidad necesaria de otras deberá **particionarse en etapas que puedan pausarse**, sea de manera **cooperativa** (desde la propia tarea) o por interrupción para una tarea prioritaria.
Las tareas en estos sistemas dedicados normalmente forman parte de una misma aplicación, por lo que deberán **compartir recursos** (datos, hardware) y eventualmente **sincronizarse.** Para esto un RTOS permite el paso de mensajes entre tareas, sincronización (esperas, semáforos etc) y alojamiento de recursos compartidos.
En resumen, las partes básicas de un RTOS serán:

- *Scheduler .* Programador que decide qué tarea se ejecuta en cada momento.
- Herramientas de sincronización y paso de mensajes.
- Servicios. Interfaz de abstracción de hardware.

Sistema Operativo de Tiempo Real (RTOS)

Ventajas:
Keil rl-arm getting started guide.pdf

Pp 20. Entry code – Exit code

Richard Barry – Using the FreeRTOS Real Time Kernel (pdf)
1.10 THE SCHEDULING ALGORITHM – A SUMMARY pág 42.

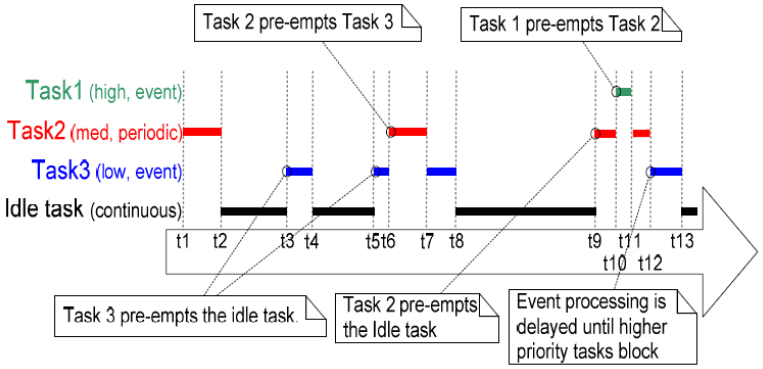


Figure 18 Execution pattern with pre-emption points highlighted

Sistema Operativo de Tiempo Real (RTOS)

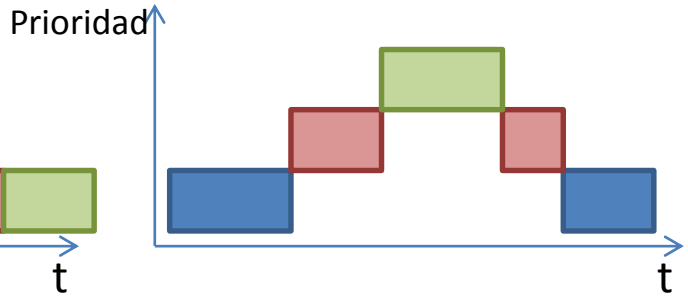
Scheduler

Está en el núcleo del RTOS. Los tres tipos fundamentales son:

Scheduling cooperativo.
Es el algoritmo más simple.
Cada tarea corre hasta que se completa y libera la CPU voluntariamente.
Una máquina de estados secuencial es un ejemplo de scheduling cooperativo.
No soporta priorización de tareas.
Una tarea puede ser demasiado larga.
El scheduling cooperativo simple, sin otras herramientas, no satisface las necesidades de tiempo real.

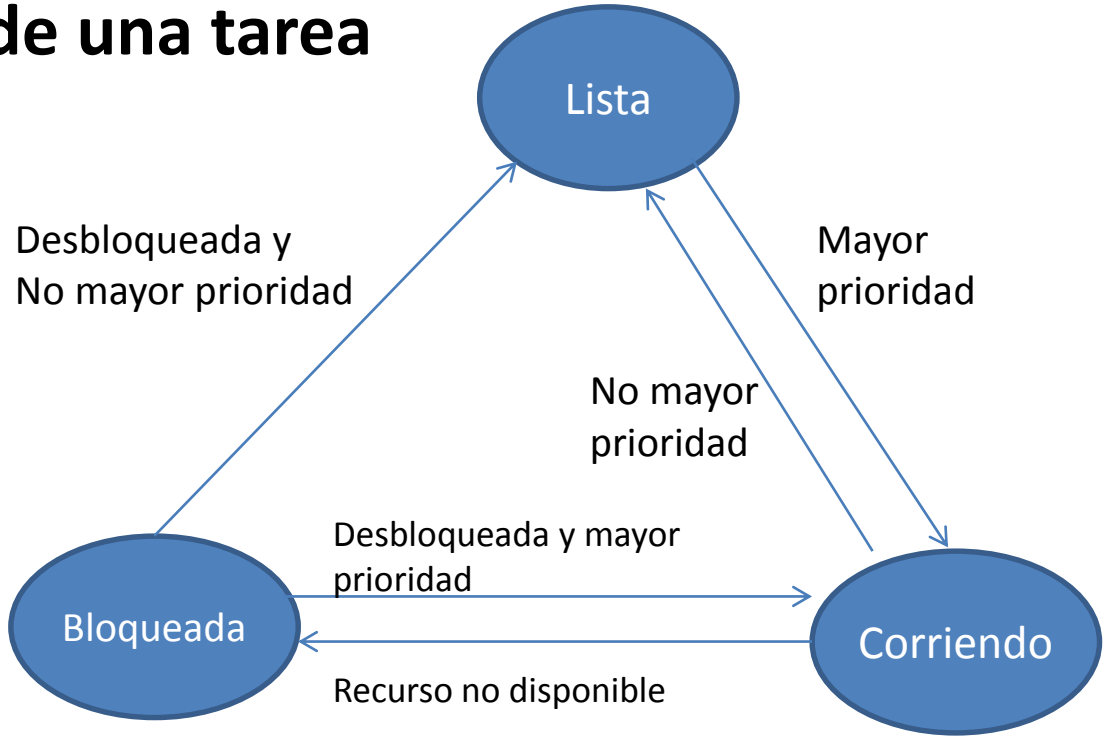
Round Robin
Es un modelo teórico sin aplicación real en forma pura.
El scheduler asigna igual tiempo a cada tarea

Pre-emptive scheduling
Es de tiempo real
Asigna prioridades a las tareas, siendo 0 la más alta.
Si hubiera tareas de igual prioridad, se puede utilizar el sistema round-robin dentro de ese nivel.
En otros sistemas se corre la tarea hasta completar antes de iniciar la otra tarea de igual nivel.



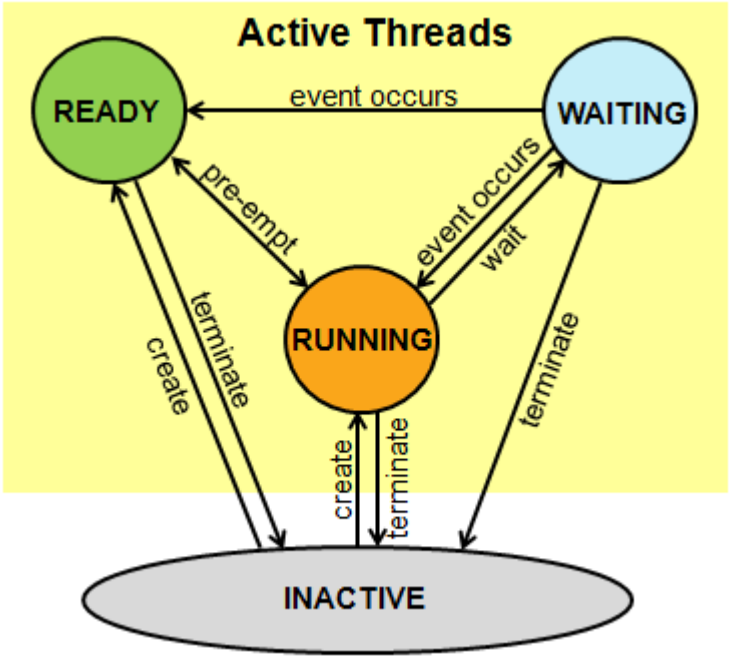
Sistema Operativo de Tiempo Real (RTOS)

Estados de una tarea



El kernel puede crear una tarea, eliminarla, cambiarle el nivel de prioridad o cambiar su estado (Lista, corriendo, bloqueada)

Estados de una tarea



Sistema Operativo de Tiempo Real (RTOS)

Servicios más comunes

- Manejo de interrupciones
- Temporización
- Manejo de dispositivos
- Manejo de memoria

Herramientas de sincronización y paso de mensajes

- **Semáforos:** Tipo de flag para arbitrar acceso a recursos compartidos
- **Flags de eventos:** Para sincronizar actividad entre tareas.
- **Colas (queues):** Para enviar mensajes entre tareas.

Sistema Operativo de Tiempo Real (RTOS)

RTOS del compilador CCS

Es un tipo de RTOS estático colaborativo, con un sistema de intervalos precalculados de acuerdo a lo que ha definido el programador.
Utiliza uno de los timers disponibles en el microcontrolador.
No utiliza interrupciones.
Dispone de funciones que permiten particionar tareas, sincronizar, pasar mensajes, calcular tiempos de ejecución etc.

Sistema Operativo de Tiempo Real (RTOS)

RTOS del compilador CCS.

Funciones de gestión de tareas

- ***rtos_run():*** Inicia la operación del RTOS. Todas las funciones de control de tareas se implementan luego de invocar esta función.
- ***rtos_terminate():*** Termina la operación del RTOS. El control vuelve al programa original, sin RTOS.
- ***rtos_enable(***tarea***):*** Recibe el nombre de la tarea como argumento y la habilita de manera que la función *rtos_run()* puede llamarla de acuerdo al scheduling.
- ***rtos_disable(***tarea***):*** La funcion inhibe la tarea de forma que no puede ser llamada por la función *rtos_run()*, hasta ser eventualmente re-habilitada con *rtos_enable()*
- ***rtos_ yield():*** Cuando es llamada desde una tarea devuelve el control a *rtos_run*. En la próxima llamada continuará ejecutándose desde el punto posterior a rtos_yield(). Puede servir para particionar una tarea larga en etapas más cortas.

Sistema Operativo de Tiempo Real (RTOS)

RTOS del compilador CCS.

Funciones de sincronización y paso de mensajes

• ***rtos_msg_send(***tarea,char dato***) :***Envía el byte **dato** a la **tarea** especificada. El dato es colocado en la cola de la tarea.

• dato=***rtos_msg_read():*** lee el dato tipo byte localizado en la cola de la tarea que invoca esta función**.**

• n=***rtos_msg_ poll() :*** indica cuántos caracteres hay en la cola. Esta función debería siempre ser llamada antes de rtos_msg_read().

• ***rtos_wait(***sem***)***: Recibe como argumento el nombre de un semáforo, esto es una variable que valdrá 0 cuando el recurso no esté disponible ó >0 cuando el recurso esté disponible. Si el recurso asociado al semáforo no está disponible entrega el control al rtos (idem *rtos_yield*). Si está disponible, lo decrementa y toma el recurso correspondiente.

• ***rtos_signal(***sem***)*** : Incrementa al semáforo **sem** cuando libera un recurso compartido.

• ***rtos_await(***expr***)*** : Espera (es decir devuelve el control al rtos) hasta que la expresión booleana **expr** sea verdadera.

Sistema Operativo de Tiempo Real (RTOS)

RTOS del compilador CCS.

Funciones de diagnóstico de tiempos

• ***rtos_overrun(***tarea***):*** Devuelve TRUE si **tarea** ha excedido su tiempo asignado.

• ***rtos_stats(***tarea,&stat***)***: Devuelve los siguientes indicadores "estadísticos"sobre una tarea específica: **uint32** total_ticks, **uint16** min_ticks, **uint16** max_ticks, **uint16** hns_per_tick.

total_ticks: Duración total de la tarea, en "ticks".
min_ticks: Duración mínima de la tarea en "ticks".
max_ticks: Duración máxima de la tarea en "ticks".
hns_per_tick: Permite calcular el tiempo a partir de los ticks, según la fórmula

$$\mu s = (ticks \cdot hns\_per\_tick) / 10$$

Estas funciones necesitan la opción *statistics* en la directiva #use rtos ….

Sistema Operativo de Tiempo Real (RTOS)

RTOS del compilador CCS.

Directiva de inicialización del RTOS

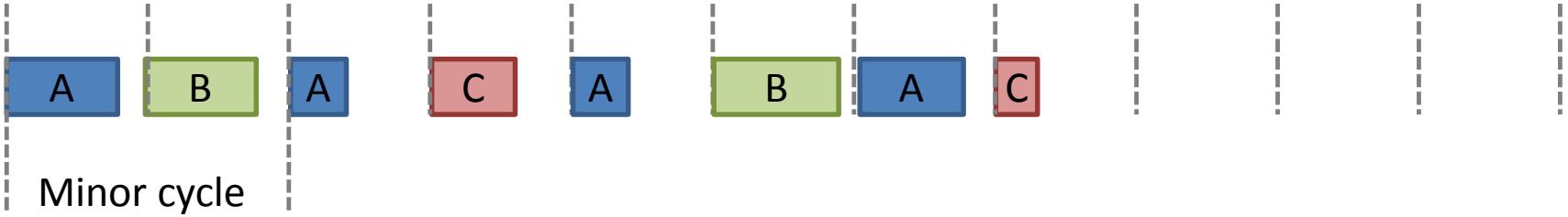
Se coloca arriba, en la zona de declaración de variables globales.

```
#use rtos(timer=x,[minor_cycle=cycle_time],[statistics])
```

timer: cualquiera disponible
minor_cycle: Tiempo de repetición de la tarea más frecuente, en ms, us, ns.

Ejemplo:

```
#use rtos(timer=1, minor_cycle=1ms)
```



Sistema Operativo de Tiempo Real (RTOS)

RTOS del compilador CCS.

Directiva de inicialización de cada tarea

Se coloca sobre la rutina que será la tarea

#task(**rate**=xxxx, [**max**=yyyy], [**queue**=z])

rate: intervalo en el cual será llamada. En ms, us etc. **Debe ser múltiplo del minor cycle** especificado en **#use rtos**

max: Tiempo estimado del procesamiento más lento de la tarea. Debe ser menor o igual al **minor cycle.**
queue: tamaño del buffer para intercambio de mensajes.

Ejemplo:

```
#task(rate=40ms, max=5ms)
void funcion()
    {
    }
```

RTOS del compilador CCS.

Ejemplo 1: Lectura de encoder, cálculo y transmisión

```
#use fast_io (D)

#use rtos(timer=1,minor_cycle=10 ms)


char comando[10];
int indCom;
unsigned int16 PulsosEncoder;
signed int16 posActual,posAnterior;
signed int16 velActual;
int1 EstadoT;

#task(rate=10ms,max=2ms)
void Calcula ( )
{
      output_toggle(PIN_B0);
      posAnterior=posActual;
      posActual=POSCNT;
      velActual=(posActual-posAnterior)*100;
}

#task(rate=20ms,max=8ms)
void Transmite()
{
     printf("%ld,%ld\r",posActual>>2,velActual);
}
```

```
void main()
{
   set_tris_a(0xFF);
   set_tris_d(0xFE);
   set_tris_b(0xF0);
   output_b(0);
   setup_uart(38400);
   EstadoT=0;
   PulsosEncoder=65535;
//   QEICON.QEIM20=QEIx4MatchR
   QEICON.QEIM20=QEIx4IndexRes


   DFLTCON=0x7A;
   MAXCNT=PulsosEncoder;

   printf("OK\r\n");

   enable_interrupts(INT_RDA);
   enable_interrupts(GLOBAL);

   rtos_run();

}
```

```
void interpretaComando()
{
   switch(comando[0])
   {
      case 'T':rtos_enable(Transmite);
               break;
      case 't':rtos_disable(Transmite);
               break;
      case 'R':POSCNT=0; //POSCNTH=0;POSCNTL=0;
               break;
      case 'X':
               printf("X=%ld\r",posActual>>2);
               break;
      case 'V':
               printf("V=%ld\r",velActual);
               break;
      case 'E':rtos_enable(Calcula);
               break;
      case 'D':rtos_disable(Calcula);
               break;
      default:
               break;
   }
}
#int_RDA
void  RDA_isr(void)
{
   char temp;
   temp=getc();
   switch(temp)
   {
      case ':':indCom=0;break;
      case 13:interpretaComando();break;
      default:comando[indCom]=temp;indCom++;break;
      return;
   }
}
```

RTOS del compilador CCS.

Ejemplo 2: Idem 1, con estadísticas

```
#use fast_io (B)
#use fast_io (D)

#use rtos(timer=1,minor_cycle=10 ms,statistics)


char comando[10];
int indCom;
unsigned int16 PulsosEncoder;
signed int16 posActual,posAnterior;
signed int16 velActual;
int1 EstadoT;

#task(rate=10ms,max=2ms)
void Calcula ( )
{
      output_toggle(PIN_B0);
      posAnterior=posActual;
      posActual=POSCNT;
      velActual=(posActual-posAnterior)*100;
}

#task(rate=20ms,max=8ms)
void Transmite()
{
     printf("%ld,%ld\r",posActual>>2,velActual);
}
```

```
void main()
{
   set_tris_a(0xFF);
   set_tris_d(0xFE);
   set_tris_b(0xF0);
   output_b(0);
   setup_uart(38400);
   EstadoT=0;
   PulsosEncoder=65535;
//   QEICON.QEIM20=QEIx4MatchRes;
   QEICON.QEIM20=QEIx4IndexRes; //


   DFLTCON=0x7A;
   MAXCNT=PulsosEncoder;

   printf("OK\r\n");

   enable_interrupts(INT_RDA);
   enable_interrupts(GLOBAL);

   rtos_run();

}
```

```
void interpretaComando()
{
 struct rtos_stas_struct {
      unsigned int32 total_ticks;   //number of ticks the task has
      unsigned int16 min_ticks;   //the minimum number of ticks
      unsigned int16 max_ticks;  //the maximum number of ticks
      unsigned int16 hns_tick;       //us = (ticks*hns_per_tick)/10
 } MisStats;
 unsigned int16 Ktick;
   switch(comando[0])
   {  case 'T':rtos_enable(Transmite);
               break;
      case 't':rtos_disable(Transmite);
               break;
      case 'R':POSCNT=0; //POSCNTH=0;POSCNTL=0;          //pone a 0 encoder
               break;
      case 'X':printf("X=%ld\r",posActual>>2);
               break;
      case 'V':printf("V=%ld\r",velActual);
               break;
      case 'E':rtos_enable(Calcula);
               break;
      case 'D':rtos_disable(Calcula);
               break;
      case 'S':switch (comando[1])
                  {
                  case 'T':rtos_stats(Transmite,&MisStats);
                  printf("Transmite\r\n");
                  break;
                  case 'C':rtos_stats(Calcula,&MisStats);
                  printf("Calcula\r\n");
                  break;
                  }                    //microsegundos=(ticks*hns_per_tick)/10
               Ktick=MisStats.hns_tick;
               MisStats.total_ticks = MisStats.total_ticks * Ktick /10;
               MisStats.max_ticks = MisStats.max_ticks * Ktick /10;
               MisStats.min_ticks = MisStats.min_ticks * Ktick /10;
               printf("Ttotal:%lu us\r\nTmin:%lu us\r\nTmax:%lu us\r\n",
                    MisStats.total_ticks,MisStats.min_ticks,MisStats.max_ticks);
               break;
      default: break;
   }
}
```