Interfaz de Computadores

https://github.com/AdriCri22/Computer-Interfacing-CI-FIB

Tema 1 y 2 – Introducción a la arquitectura PIC18

Memory

- Mapped I/O: cualquier instrucción puede leer/escribir por un puerto de entrada/salida y escribir/leer en la memoria, por lo que hay que especificar que modulo es con el que nos comunicamos.
- Isolated I/O: Hay un bus de control para acceder por separado los puertos I/O y la memoria del dispositivo, por lo que usamos instrucciones separadas para comunicarnos.

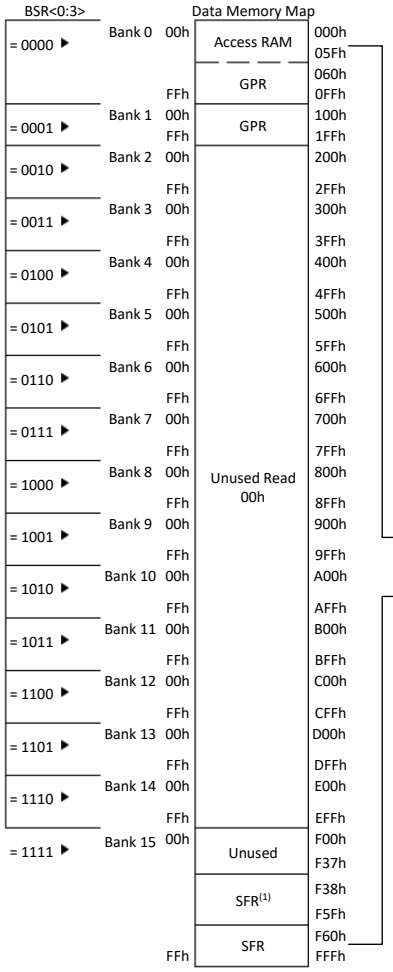
RISC (Reduced Instruction Set Computing): Se usa para programas más simples, ya que tarda menos en programarse y depurarse.
CISC (Complex Instruction Set Computer): Se usa para programas con operaciones más complejas

PIC18 Architecture – Data/Program buses

Bus de memoria del programa	
21 bits	dirección del programa
16 bits	bus de instrucciones

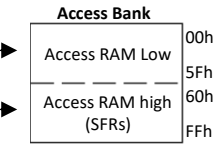
Bus de memoria de datos	
12 bits	dirección de registro
8 bits	bus de datos

Si preguntan por el tamaño → 2^x



Dado que la mayoría de las instrucciones usan 8 bits para especificar la dirección del registro, se divide la memoria en 16 bancos de 256 bytes (la memoria entera tiene 4096 bytes), de esta manera sólo un banco puede estar activo, este banco se selecciona mediante el registro **BSR**, pero dado que al cambiar de banco es muy frecuente que se produzcan errores se usa el **Access Bank**: es un banco de 256 bytes donde los primeros 96 bytes están dedicados a GPR y los 160 bytes restantes a SFR

- GPR (general-purpose registers) → Guardan los datos, hay 6 bancos GPR cada banco contiene 256 direcciones
- SFR (special-function registers) → Especifican la operación del microcontrolador hay 1 banco SFR



Mem. Pos. $0x321$
(8 bits) indica la
(4 bits del BSR) indica el banco

ADDWF f, d, a → Access Bank o
Donde se almacena el
Registro del que cogemos el

SI "a" = 0 – A → Se accede a la access bank y se ignora el BSR
Si "a" = 1 – BANKED → Se accede al registro BSR donde se especifica el banco al que se quiere acceder

Si "d" = 0 – W → Se almacena en el registro W(WREG)
Si "d" = 1 – F → Se almacena en el registro origen

Indirect adressing

- FSRx (File Select Registers) → Apunta a los 12 bits de la dirección con la que se quiere trabajar, para indicar la dirección se escribe en par FSRxH (High - 4/8 bits) y FSRxL (Low - 8/8 bits).
- INDFx → Indica la dirección del FSRx.
- POSTDEC, POSTINC, PREINC, PLUSW → Indican la dirección del FSRx y luego decrementa, aumenta o cambia a otra dirección el FSRx, después de hacer la operación.

Pipelining

$$F_{cy} = \frac{F_{OSC} \rightarrow Frec.clock}{4} \rightarrow T_{cy} = \frac{1}{F_{cy}} = \frac{4}{F_{OSC}}$$

#define <name> [<string>] → sustituye <string> por <name> en código ensamblador.
#include <include_file> → incluye un archivo de origen.
[<label>] org <expr> → Esta directiva establece el origen del programa para el código subsiguiente en la dirección definida en <expr>.

Tema 3 – Puertos entrada/salida

Esta es la manera en la que el microcontrolador se comunica con otros periféricos:

I/O Port Structure	Data Register: para el paso de datos Control Register: Retiene los comandos del procesador al puerto Status Register: Monitorea la actividad de entrada/salida

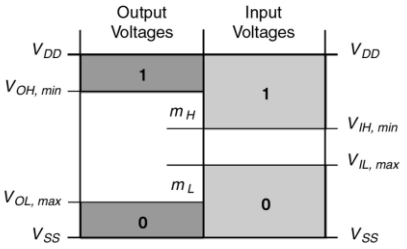
Un puerto es un grupo de 8 pines donde:

TRISx	Data direction register	1 Input 0 Output
LATx	Latch register	Lee lo que hemos escrito en el puerto y puede escribir en el puerto de salida
PORTx	Data register	Lee lo que hay en la entrada y escribe en el puerto de salida
ANSELx	Analog input control	0 Digital 1 Analog

Un pin de salida da corriente si está a "1" y chupa si está a "0"
Un pin de entrada chupa corriente si está a "1" y da si está a "0"

Los pines RB<2:0> se pueden usar como interrupciones externas (INT0, INT1 y INT2)

Hay que considerar los **niveles de voltaje** de las entradas y salidas, son los rangos de voltaje en los que se detecta un input o un output:
V_{OL} = Tensión de salida a nivel bajo
V_{OH} = Tensión de salida a nivel alto
V_{IL} = Máxima tensión de entrada que será interpretada como "0"
V_{IH} = Mínima tensión de entrada que será interpretada como "1"



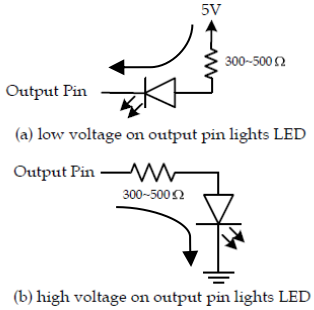
La corriente se usa para determinar el ***static fanout of a device*** (el número de inputs que pueden ser conectados en un output preservando los voltajes requeridos)

Static fanout for a low output: $n_L = \frac{|I_{OL,max}|}{|I_{IN}|}$
Static fanout for a high output: $n_H = \frac{|I_{OH,max}|}{|I_{IN}|}$
$n = \min[n_H, n_L]$

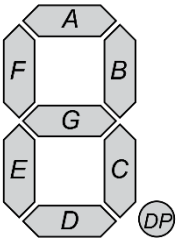
I_{OH} = Corriente que fluye fuera de un HO
I_{OL} = Corriente que fluye fuera de un LO
I_{IN} = Corriente que fluye fuera o dentro de un pin de entrada

Interfacing with LEDs

$V_{que\ le\ damos} = V_{que\ ha\ de\ pasar\ por\ el\ LED} + I_R \times R$



7 Segment Displays



Ánodo común: enviando un "0" se ilumina el segmento.

Cátodo común: enviando un "1" Se ilumina el segmento.

PORTx

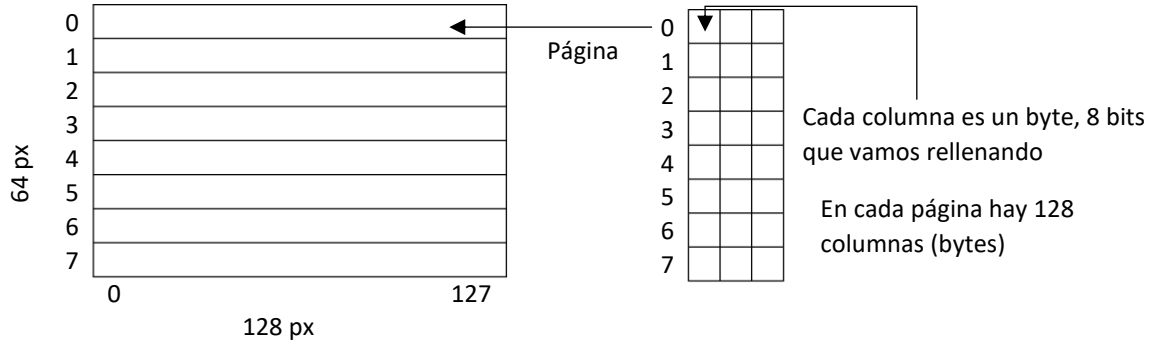
x7	x6	x5	x4	x3	x2	x1	x0
DP	G	F	E	D	C	B	A

Interfacing with Keypad Tipos de interrupciones de llave:

1. Membrana: Una membrana de plástico o goma que presiona un conductor sobre otro
2. Capacitivo: Dos placas paralelas que al presionar cambia la distancia y por lo tanto la capacitancia.
3. Efecto Hall: Flujo magnético creado por un imán perpendicular a un cristal, cuando el imán se mueve se produce un campo eléctrico.
4. Mecánico: Dos contactos metálicos se unen para completar un sistema eléctrico.
    - El más barato y utilizado

Las teclas se distribuyen en filas y columnas (simples conductores) del teclado, cuando se pulsa una tecla, la fila y la columna correspondientes se ponen en cortocircuito y se detecta low. Como el voltaje de la tecla cae y sube varias veces hasta estabilizarse un ***debouncer*** reconoce cuando una tecla ha sido pulsada después de ciertos ms. Hay **diversas técnicas** para tratar estos rebotes **tanto des del hardware como programando** (una técnica es el **wait-and-see**).

GLCD



Tema 4 – Interrupciones

El microprocesador hace unas transferencias de datos sumamente rápidas y los dispositivos de entrada salida no aceptan esa velocidad por lo que para estar seguro de que el periférico esta preparado para recibir un dato. Hay varias maneras de solucionar esto:

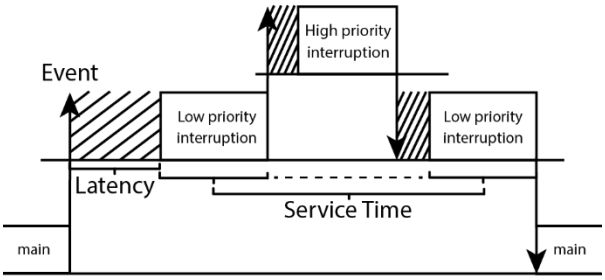
- **Blind Cycle**: Se envía un dato y se espera una cantidad de tiempo y se da por echo que se ha recibido (cuando sabemos que el tiempo de respuesta del dispositivo es corto).
- **Gadfly**: Hay un bucle que comprueba el estado del dispositivo (cuando el tiempo real de respuesta no es crucial).
- **Interrupciones**: Se usa el hardware para causar una ejecución especial del software (cuando el tiempo de respuesta es crucial).
- **Periodic Polling**: Se usa el reloj para periódicamente comprobar el estado del dispositivo (cuando se requieren interrupciones, pero el dispositivo no soporta solicitudes).
- **DMA**: Transferencia de datos directa y en tiempo real por un canal entre el dispositivo y la memoria sin pasar por el procesador (cuando el ancho de banda y la latencia son importantes).

Interrupciones → Mientras se va ejecutando el programa principal, si ocurre la interrupción que nosotros hayamos puesto, entonces el *program counter* cambia para acceder a la función de interrupción (esta función no ha de estar en programa principal), ejecuta la interrupción que hayamos programado y vuelve donde estaba el *program counter* antes de la interrupción. Soporte para interrupciones:

- Guarda el PC para el return (`retfie`).
  Guarda otros registros especiales (STATUS, WREG, BSR), solo en high priority, `retfie FAST` → Devuelve todo a su sitio.
- Define la ubicación de donde ir cuando ocurre la interrupción.
- Determina la causa de la interrupción.

Existen 2 tipos de interrupciones (Si un evento de cierta prioridad ocurre no puede venir un evento de la misma prioridad y interrumpirlo):

- De baja prioridad (dirección 0x18)
- De alta prioridad → Cuando una interrupción de baja prioridad está en curso y ocurre otra interrupción de más importancia, entonces esta interrumpe la de baja para atender la de alta prioridad. (dirección 0x08)



Parámetros

- Prioridad → Que interrupción se atiende primero.
- Enmascarar → Que hacer con la interrupción.
- Latencia → Cuanto se tarda en atender la interrupción (hay un pequeño delay).
- Service Time → Cuanto tarda la interrupción.

Registros:

- **RCON** (Reset Control Register) → **IPEN** → 1 Enable priority levels (sino todas son high priority)
- **INTCON** (Interruption Control Register) → **GIEH** → 1 Enable high priority, **GIEL** → 1 Enable low priority
- **INTCON1**
- **INTCON2**
- **IE** (interruption enable) bit → 0 Disabled, 1 Enabled
- **IP** (priority interruption) bit → 0 low priority, 1 high priority
- **IF** (interruption flag) bit → 1 event happened, otherwise 0. (Cuando ocurre la interrupción hay que dejarlo a 0 manualmente)

Interrupciones:

- INT0…INT2 (RB0…RB2) → Interrupciones del pin por flancos
  - INTEDGx → 0 Falling or 1 rising edge
- TMR0 → Interrupción por overflow

High priority

```
void interrupt FUNCION(void) {
   if (INTERRUPCIONIE && INTERRUPCIONIF) {
      …
   }
}
```

Low priority

```
void interrupt low_priority FUNCION(void) {
   if (INTERRUPCIONIE && INTERRUPCIONIF) {
      …
   }
}
```

Tema 5 – Timers CCP Module

Un **Timer** es un registro que su valor va creciendo de manera constante sin el uso de la CPU.

Configurar tiempo del Timer:

1- $T_{cy} = \frac{1}{F_{cy}} = \frac{4}{F_{OSC}} = \frac{4}{8\ MHz\ (frec.PIC18F45K22)} = 0{,}5\ \mu s$

2- $0{,}5\ \mu s * prescaler\ (el\ clock\ va\ x\ veces\ más\ lento) * nTics = tiempo$

3- En el código escribir: $TMRx = 0x0000 - nTics$ (si el TMR está a 16 bits)
$TMRx = 0x00 - nTics$ (si TMR está a 8 bits)

Es mejor configurarlo utilizando $TMRxH$ (bits de más peso) y $TMRxL$ (bits de menos peso) porqué al usar 16 bits realmente lo que hace es guardarlo en 2 de 8 bits.

El Timer genera la interrupción al pasar de 0xFFFF a 0x0000 (al hacer overflow) si está habilitado.

- Timer 0 puede ser de 16 bits o de 8 bits
  - RCONbits.IPEN = 1 Activa niveles de interrupciones, 0 desactiva estos.
  - INTCONbits.GIEH = 1 Activa high priority interrupts, 0 Desactiva todas las interrupciones (Las low priority interrupts también)
  - INTCONbits.GIEL = 1 Habilita low priority interrupts, 0 Deshabilita las low priority interrupts
  - INTCONbits.TMR0IE = 1 Habilita TMR0 overflow, 0 Deshabilita TMR0 overflow
  - INTCONbits.TMR0IF = 1 se ha producido TMR0 overflow, 0 no se ha producido
  - Configurar también el registro T0CON

- Timer 1, Timer 3 y Timer 5 es de 16 bits dependiendo de la fuente del reloj.
  - Se configura usando $TxCON$ y $TxGCON$.
  - Se pueden usar para delays y medir la frecuencia de una señal desconocida (con módulos CCP).
- Timer 2, timer 4 y Timer 6 son de 8 bits.
  - Se configuran usando $TxCON$ y PRx.
  - Se pueden usar como fuente de señales PWM (módulos CCP).

```
// Ejemplo de configuración del Timer 0
void configPIC() {
   ANSELC = 0;              // PORTC configured as Digital
   PORTC = 0;               // Set port values
   TRISC = 0xFF;           // Configure C as Input

   T0CONbits.TMR0ON = 1;    // Enables Timer0
   T0CONbits.T08BIT = 0;    // Configure Timer0 as a 16-bits timer
   T0CONbits.T0CS = 0;      // Instruction cycle clock
   T0CONbits.PSA = 0;       // Prescaler is assigned
   T0CONbits.T0PS = 0b101;  // Timer0 prescaler select bits
   INTCONbits.GIE = 1;      // Enable Global interrupts

}
```

```
// ESPERA PASIVA -> Usando interrupciones
void interrupt highRSI(void) {
   if (INTCONbits.TMR0IE && INTCONbits.TMR0IF) {
      INTCONbits.TMR0IF = 0;
      // Código después de la espera
      ...
   }
}

void main() {
  configPIC();
  while (1) {
      ...
      TMR0H = 0x00 - nTicksH; // Parte alta
      TMR0L = 0x00 - nTicksL; // Parte baja
      ...
  }
}
```

```
// ESPERA ACTIVA
// El Pic no hace nada más en ese
// tiempo (se queda como paralizado)
void main() {
  configPIC();
  while (1) {
      ...
      TMR0 = 0x0000 - nTicks;
      INTCONbits.TMR0IF = 0;
      while (!INTCONbits.TMR0IF);
      // Código después de la espera
      ...
  }
}
```

CCP (Capture, Compare and PWM (Pulse Width Modulation))

- **Capture**: Guarda el tiempo del Timer (1, 3 o 5) asociado a un periférico CCP cada vez que viene un flanco. Una vez iniciado el tiempo va corriendo hasta que detecta un flanco ascendiente o descendiente (dependiendo de la configuración) de una señal externa (por un pin).

$$nTics = CCPRx - anterior$$
$$anterior = CCPRx$$

- **Compare**: Va comparando el tiempo del registro CCPR con el Timer (1, 3 o 5) cada ciclo de reloj. Cuando estos 2 registros son iguales → podemos poner a 1, 0 o un toggled (si estaba a 1 pasa a 0 y al revés).

$$CCPRx = CCPRx + nTicsSemiperiodo$$

- **PWM**: Genera ondas especificando su frecuencia y su duty cycle $d_c = \frac{tiempo\ que\ está\ en\ 1\ la\ onda}{tiempo\ total} \cdot 100$ utilizando los Timers 2, 4 o 6.
  - Cuando $TMRx = PRx$ → Pasa de 0 a 1 (Genera un flanco ascendente, reset de la señal de salida).
  - Cuando $TMRx = CCPRxL$ → Pasa de 1 a 0 (Genera un flanco descendente).
  - Dado que los Timers para especificar el duty cycle tienen 256 niveles, el fabricante nos proporciona 2 bits más de menos peso, por lo que nos deja 10 bits y por lo tanto nos deja más resolución para especificar la interrupción.

Hay 5 Módulos CCP cada uno hay que asociarlo a un Timer (dependiendo de que queremos hacer con la CCP)

- Para configurarlo se usan los registros $CCPxCON$ (para seleccionar el modo), $CCPTMRS0$ y $CCPTMRS1$ (para seleccionar el Timer usado) y $CCPRx$ ($CCPRxH$ y $CCPRxL$)
- Para las operaciones CCP hay pines CCP ($CCPx$) asociados a los RXx (hay que mirarlo en el datasheet).

```
// Ejemplo de configuración del modo PWM
void configPic () {
   TRISCbits.RC2 = 0;   // Configuramos RC2 como output

   T2CONbits.T2OUTPS = 0b0000;  // Postscaler a 0
   T2CONbits.TMR2ON = 1;        // Timer 2 operativo
   T2CONbits.T2CKPS = 0b11;     // Prescaler a 16

   // CCP1 corresponde al puerto RC2 -> configuramos CCP1 al timer 2
   CCPTMRS0bits.C1TSEL = 0b00;

   // Configurar los 2 bits extra que nos ofrecen los fabricantes para más resolución
   CCP1CONbits.DC1B = 0b00;
   CCP1CONbits.CCP1M = 0b1100;   // CCP1 configurado con el modo PWM

   INTCONbits.GIE = 1;     // Habilitamos las interrupciones globales
   PIE1bits.TMR2IE = 1;    // Habilitamos las interrupciones TMR2
   INTCONbits.PEIE = 1;    // Enables all peripherical interrupts

   // 1 ms / (0.5 us * 16 prescaler) = 125 tics
   PR2 = 125;      // Configuramos el flanco ascendente
   CCPR1L = 63;    // Configuramos el flanco descendente a la mitad del PR2
}
```

Tema 6 – Interfaces analógicas

ADC (Analog to Digital Conversion): representa un valor físico que constantemente va variando por una secuencia de valores numéricos.

Un sensor transforma un valor real en un voltaje.

- Resolución de ADC $= (V_{ref+} - V_{ref-})/2^N$ → Obtiene el valor en voltaje de da bit.
- Bits necesarios para cierta resolución $\log_2\left((V_{ref+} - V_{ref-}) \,/\, resolución\ necesitada\right)$
- $Dout = round\left[(2^N - 1) \cdot (V_{IN} - V_{ref-})/(V_{ref+} - V_{ref-})\right]$ → Convierte un voltaje en un valor binario.

Siendo $V_{ref+} = V_{DD} = 5\ V$, $V_{ref-} = V_{SS} = 0\ V$, V_{IN} el voltaje de entrada y N siendo el número de bits.

Básicamente el ADC tarda un tiempo desde que comienza hasta que acaba la conversión, el tiempo de adquisición se utiliza para cargar un condensador pequeñito que hay en la entrada del ADC.

- Si necesitas que este tiempo sea muy pequeño, bajas el ADCON2bits.ACQT, pero no nos asegura que el condensador se cargue, por lo que la conversión podría ser errónea.
- Si no tienes restricción de tiempo, subes el ADCON2bits.ACQT, si este tiempo es muy grande, el condensador estará perfectamente cargado.
- Si bajas demasiado el ADCON2bits.ACQT entonces puede ser que la lectura sea errónea. Para seleccionar este valor, hay que tener en cuenta si la señal que se va a convertir de analógico a digital es muy rápida o muy lenta. Si necesito convertir una señal de audio, necesitaré que el tiempo de conversión sea muy pequeño o estaré perdiendo parte de la señal. Si la señal es de un potenciómetro, o de un sensor de temperatura, el tiempo de conversión puede ser más largo porque la señal cambia muy lentamente y no necesito tanta velocidad de conversión.

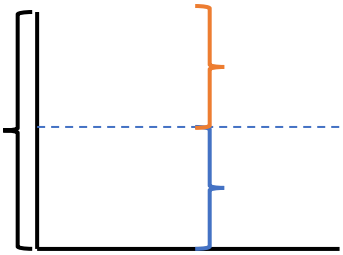
Adquisition time = valor que le damos al registro ACQS + 11 T_{AD} para la conversión ($T_{CNV} = 12$), pasan 11 ciclos, realmente pasan 12 ciclos, ya que es al siguiente cuando se produce la conversión.

En el caso de la PIC18 hay que cumplir:

$$1\mu s \leq T_{AD} = n/FOSC\ (ADCS < 2{:}0 >)\ \leq 25\mu s$$

$$7.45\mu s \leq T_{ACQ}$$

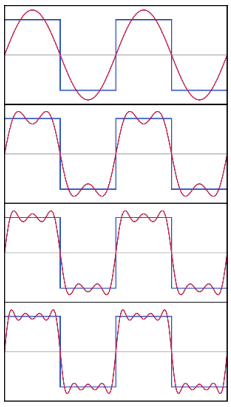
Una manera para **mejorar la resolución manteniendo los bits** es:



Haciendo que haya dos rangos de voltaje uno superior (Ej: $V_{ref+} = 5V$ y $V_{ref-} = 2{,}5V$) y otro inferior (Ej: $V_{ref+} = 2{,}5V$ y $V_{ref-} = 0V$) obteniendo así el doble de resolución.

Para saber el tiempo que hay que esperar para obtener una muestra ADC: Cantidad de información en bits por segundo $= (1 \,/\, Ts) * N\ bits\ ADC$, dónde Ts es el periodo de muestreo.

Criterio de Nyquist $f_{muestreo} > 2 \cdot f_{frecuencia\ máxima\ de\ señal}$



$F_{min\ muestreo} = 2 \cdot F_{max}$ $\quad$ $F_{\max muestreo} = \frac{FOSC/n}{ACQ+CONV}$ dónde $(n = 64/16/4)$

La serie de Fourier $x(t) = 4/\pi\ (\sin(w_0 t) + 1/3\ \sin(3w_0 t) + 1/5\ \sin(5w_0 t) + \cdots)$, va haciendo que una onda sinusoidal o cosinusoidal se parezca más a una onda cuadrada a medida que se añaden más términos a la fórmula.

Filter Anti-aliasing → Rebajan el máximo componente frecuencial (elimina los componentes de altas frecuencias, quedándose con todas las frecuencias por debajo de la frecuencia de corte). Esto soluciona el problema del criterio de Nyquist cuando la frecuencia de muestreo no cumple el requisito ($> 2 \cdot f_{freq\ \max señal}$). Haciendo que $f_{CLK} > 2 \cdot f_{cutoff}$ (Lo que hace este filtro es evitar cambios bruscos en señales de ondas, como las cuadradas dónde el cambio de 1 a 0 es brusco).

DAC (Digital to Analog Converter): Representa un valor binario a uno físico.

- $V_{out} = V_{ref-} + (V_{ref+} - V_{ref-}) * Din\ /\ (2^N - 1)$ → Pasa los bits a voltaje. Din es un número de binario.

```
// Ejemplo de cómo obtener el valor de analógico a digital por polling
int obtain_value() {
  ADCON0bits.GO = 1;
  while (ADCON0bits.GO);    // Wait until obtain the value
  int an0 = ADRESH & 0x03;  // Ignore bits 10:15
  an0 = an0 << 8;           // Move 8 bits to the left (shift left)
  an0 = an0 | ADRESL;       // Concantenate ADRESH and ADRESL
  return an0;
}
```

```
void interrupt highRSI(void) {
  // ADC por interrupciones
  if (PIE1bits.ADIE && PIR1bits.ADIF) {
    ...
  }
}
```

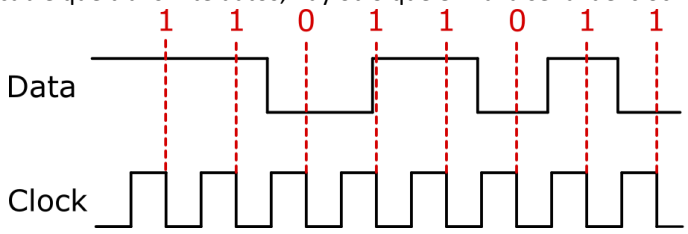
```
// Ejemplo de configuración para pasar de analógico a digital
void configPIC() {
   ADCON2bits.ADFM = 1;       // Conversion Result Format select bits: Left justified
   ADCON2bits.ACQT = 0b100;   // TAD = 8
   ADCON2bits.ADCS = 0b110;   // FOSC / 64
   ADCON1bits.PVCFG = 0b00;   // A/D Vref+ = VDD
   ADCON1bits.NVCFG = 0b00;   // A/D Vref- = VSS
   TRISAbits.RA0 = 1;         // Set pin A0 as input
   ANSELA = 1;                // Set pin A0 as analogue
   ADCON0bits.CHS = 0b00000;  // Analog channel select bits
   ADCON0bits.ADON = 1;       // ADC Enabled

}
```

Tema 7 - Serial communications interfaces

Tipos de interfaces de comunicación en serie:

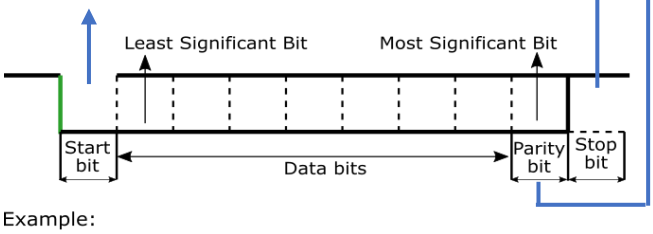
- Síncrona: Por cada cable que transmite datos, hay otro que envía la señal del clock.



  - Half-Duplex: solo existe un cable de datos, por lo que solo podemos transmitir o recibir datos al mismo tiempo, ya que si lo intentamos destrozaríamos la señal (Si por ejemplo hacemos una transmisión bidireccional de 2 bytes requerirá el doble de tiempo), se usa para n interlocutores.
  - Full-Duplex: existen 2 líneas de transmisión de datos, cada una con su clock correspondiente, por lo que podemos transmitir y recibir señales a la vez, se suele usar para 2 interlocutores.
- Asíncrona: cada extremo tiene su generador de clock configurados a la misma frecuencia para que al leer los datos sean equivalentes a cuando se transmitieron.

Es el inverso del valor lógico por defecto de la línea de transmisión, que es 0 o 1 débil

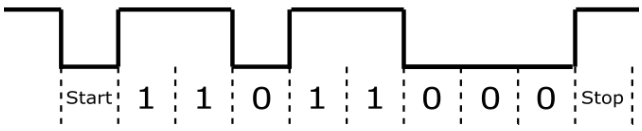
Siempre en idle status, es decir a 1



El bit de paridad es un bit extra (no es obligatorio ponerlo), sirve para indicar si el número de bits en 1 que hay es par o impar.

Dado que los datos llegan antes que el bit de paridad, el receptor puede averiguar ya el número de bits en 1 y saber si el bit de paridad será par o impar, entonces el receptor comprueba si el bit de paridad cuadra para saber si ha habido error o no en la recepción de datos.

Example:



Los bits start y stop sirven para que el receptor sincronice el clock, lo hace sincronizando el flanco descendente del clock con el flanco descendiente del start bit

  - Half-Duplex:
  - Full-Duplex

Línea serie:

Errores de transmisión:

1. Framing error: cuando ocurre un error de sincronización, es decir leemos bits que no corresponden a lo que debería. Para detectar este problema se usa el Stop bit
2. Receiver overrun: cuando el receptor lee un byte entero se guarda en un registro (TXREGx), si el programa principal deja de leer este registro el periférico informa que estamos perdiendo datos.
3. Parity errors: cuando no cuadra el bit de paridad con el cálculo que el programa ha hecho.

Registros de transmisión:

- TXREGx → Registro donde se van cargando los datos y sirve como precarga para el TSR, una vez cargado se indica en el registro TXxIF.
- TXxIF → 1 TXREGx está vacío y podemos añadir datos, si es 0 el TXREGx no está vacío y si cargamos un dato sobrescribiríamos un dato que no hemos enviado.
- TSR → registro que contiene los bits que se van desplazando hacia la derecha para salir por el cable.
- TRMT → Indica si hay algo enviándose o ya se ha enviado todo .
- Baud Rate Generator indica la frecuencia del clock
  - FOSC → Indica la frecuencia de la pic.
  - SPBRG $= n$ → Registro numérico de 8 o 16 bits
  - BRGH → 1 indica velocidad elevada, 0 velocidad baja

BR → Indica la tasa (es el dato que nos suelen dar)

$$BR = FOSC/[64 \cdot (n+1)], \qquad BR = FOSC/[16 \cdot (n+1)], \qquad BR = FOSC/[4 \cdot (n+1)]$$

Registros de recepción:

- FERR → Permite detectar el farming error
- OERR → Permite detectar el overrun error
- RCREGGx → Nos da el dato recibido, hay 2 posiciones por lo que si no hemos podido leer un dato, el receptor lo escribirá en la segunda posición, esto lo hace la FIFO de manera interna, es decir no hay que preocuparse por ello. Si dejas de leer un tercer byte da overrun error.
- RCxIF → 1 indica que se ha recibido el dato y se puede leer, 0 indica que no hay ningún dato para leer.

La Pic 18 tiene 2 módulos para comunicarse en serie:

- USART1 → RC6/TX1/CK1 y RC7/RX1/DT1
- USART2 → RD6/TX2/CK2 y RD7/RX2/DT2

```
void configTransmition() {
   ANSELC = 0;              // PORTC configured as Digital
   PORTC = 0;               // Set port value

   TRISCbits.RC7 = 1;       // Configure RX1 pin for input
   TRISCbits.RC6 = 0;       // Configure TX1 pin for output

   // Transmission configuration
   TXSTA1bits.TX9 = 0;      // 8 bits transmission
   TXSTA1bits.TXEN = 1;     // Enable transmission
   TXSTA1bits.SYNC = 0;     // Asynchronous mode
   TXSTA1bits.BRGH = 1;     // Asynchronous mode: High speed
   PIE1bits.TXIE = 0;       // Disable transmit interrupt
   RCSTA1bits.SPEN = 1;     // Enable USART port

   // Receiver configuration
   RCSTA1bits.SPEN = 1;     // Enable USART port
   RCSTA1bits.RX9 = 0;      // 8-bits reception
   RCSTA1bits.CREN = 1;     // Enables receiver

   BAUDCON1bits.BRG16 = 1;  // 8-bit Baud Rate
   // SYNC = 0; BRG16 = 1; BRGH = 1; -> 16-bit/Asynchronous -> FSOC / [4 * (n + 1)]
   // FOSC = 8MHz; Desired baud rate = 115200; multiplier = 4
   // n = ((8MHz / 115200) / 4) - 1 = 16.361 -> n = 16 (Half Round Up)
   // Bound rate = 8MHz / (4 * (16 + 1)) = 117647.0588 ->
   // -> error = (117647.0588 - 115200) / 115200 = 0.021242 = 2.1242%
   SPBRG1 = 16;
}
```

```
// Subroutine to output a character to USART1 using the polling method
void send(char c) {
   while (!PIR1bits.TX1IF);
      TXREG1 = c;
}

// Subroutine to input a character to USART1 using the polling method
char receive() {
   while (!PIR1bits.RC1IF);
      return RCREG1;
}
```

```
void interrupt highRSI(void) {
  // Input a character to USART1 using interruptions
  if (PIE1bits.RC1IE && PIR1bits.RC1IF) {
    char c = RCREG1;
  }

  // Output a character to USART1 using interruptions
  if (PIE1bits.TX1IE && PIR1bits.TX1IF) {
    char c = 'x';
    TXREG1 = c;
  }
}
```

Modulo MSSP

Se usa para el protocolo SPI o I2C, no a la vez, ya que el hardware que necesitan es muy similar.

	SPI	I2C
Serial data out (SDO)	RC5/SDO	-
Serial data in (SDI)	RC4/SDI	RC4/SDA
Serial clock (SCK)	RC3/SCK	RC3/SCL
(opcional) Pic in Slave mode (SS)	RA5/SS1	RA5/SS1

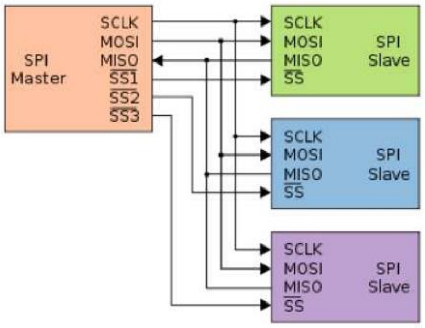
- SSPBUF registro de recepción/transmisión de datos
- SSPCON1 MSSP control register
- SSPSTAT MSSP status register

$$tiempo = \frac{bits\ transmitidos}{baud\ rate}$$

SPI (Serial Peripheral Interface Bus)

Es un enlace estándar de comunicación en serie síncrona full-duplex. Cables:

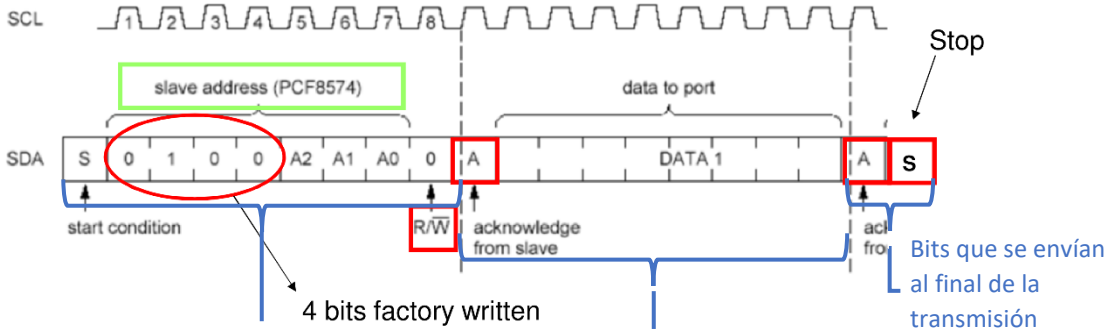
- SCLK: cable del clock.
- MOSI: cable de transmisión de datos.
- MISO: cable de recepción de datos.
- SSx (Para el dispositivo máster, x es un número): bit que nos indica con que esclavo desea el máster comunicarse, solo puede comunicarse con 1 a la vez porque están conectados tipo Bus, es decir, que están todos los esclavos conectados físicamente al mismo cable.
- SS (Para el esclavo): bit que indica si el máster desea comunicarse con él o no.



Registro SSPBUF: Al cargar un dato en este registro automáticamente se inicia la transmisión y recepción de datos, el dato que hayamos cargado pasa a cargarse en el registro SSPRS (registro que nosotros no podemos acceder), y según se va enviando el byte (por el cable SDO), se va recibiendo otro byte (por el cable SDI).

I2C, I²C (Inter-Integrated Circuit Bus)

Comunicación en serie síncrona half-duplex, dado que solo se utiliza una línea de datos (SDA), tenemos menos throughput, menos número de bits por segundo.



Se envía al principio de la transmisión

Estos son los bits que se envían en la transmisión hasta que ya no quedan más datos por enviar

Bits que se envían al final de la transmisión

A diferencia de la comunicación SPI, el I2C o recibe datos o los envía por el registro SSPRS, además si configuramos la Pic como slave, el registro SSPADD indica la dirección que le corresponde como esclavo y se nos indica si hay match, para saber si el máster quiere iniciar una transmisión de datos.

Asynchronous serial interfaces (1Wire)

Comunicación en asíncrona half-duplex, que además de transmitir información mediante el cable de recepción y transmisión de datos, subministra energía mediante ese mismo cable, además cada dispositivo 1 wire tiene un identificador único.

Si queremos enviar un 1, el master envía un 0 durante poco rato y un 1 durante más rato, y para enviar un 0, el master envía un 0 durante más rato y un 1 durante poco rato, de esta manera nos aseguramos que el condensador que tiene el dispositivo 1 wire pueda mantener cargado el dispositivo y dado que vamos generando flancos descendentes periódicamente (el 0 inicial al enviar 1 o el flanco descendente que viene de dejar un 1 al final de cada bit 0 que enviamos), nos sirve para comprobar que los datos se envían correctamente. En el caso de recibir datos el máster deja a 0 durante muy poco tiempo, y si el esclavo lo sigue dejando a 0, envía un 0 (después lo vuelve a poner a 1), y si lo cambia directamente a 1, envía un 1.

USB (Universal Serial Bus)

Comunicación en serie asíncrona half-duplex que sigue una arquitectura master-slave.

Al enviar una tira de datos que sigue una codificación NRZi, para saber que los datos se envían correctamente se comprueba mediante flancos cada x tiempo, en el caso de que se envían 1 de manera continuada, el usb especifica que cada seis unos seguidos envíe un 0.

Para iniciar una transacción el máster primero identifica con quien quiere hablar mediante una dirección + endpoint y especificara si hace un IN o un OUT de los datos, en el caso del IN enviará una petición de datos, el esclavo contestará con los datos que el máster a pedido DATA0 o DATA1, en el caso de OUT el master enviará los datos y para que el que ha transmitido de los datos, ya sea el master o el slave, sepa que han llegado correctamente, el receptor envía el **ACK**.

NACK → Indica que no se pueden recibir los datos porque o no tiene memoria o está ocupado.

Endpoints: con canales lógicos para comunicaciones, son canales de comunicaciones entre diferentes procesos. Hay de 4 tipos:

- Control: para enviar comandos para controlar ciertos periféricos (paquetes pequeños).
- Interrupciones: son los que se consultan periódicamente (paquetes pequeños).
- Bulk: para transmitir una gran cantidad de datos, pero sin requisitos de tiempo.
- Isochronous: transmisión de datos periódicos, pero sin control de errores, por ejemplo el audio en una videollamada.