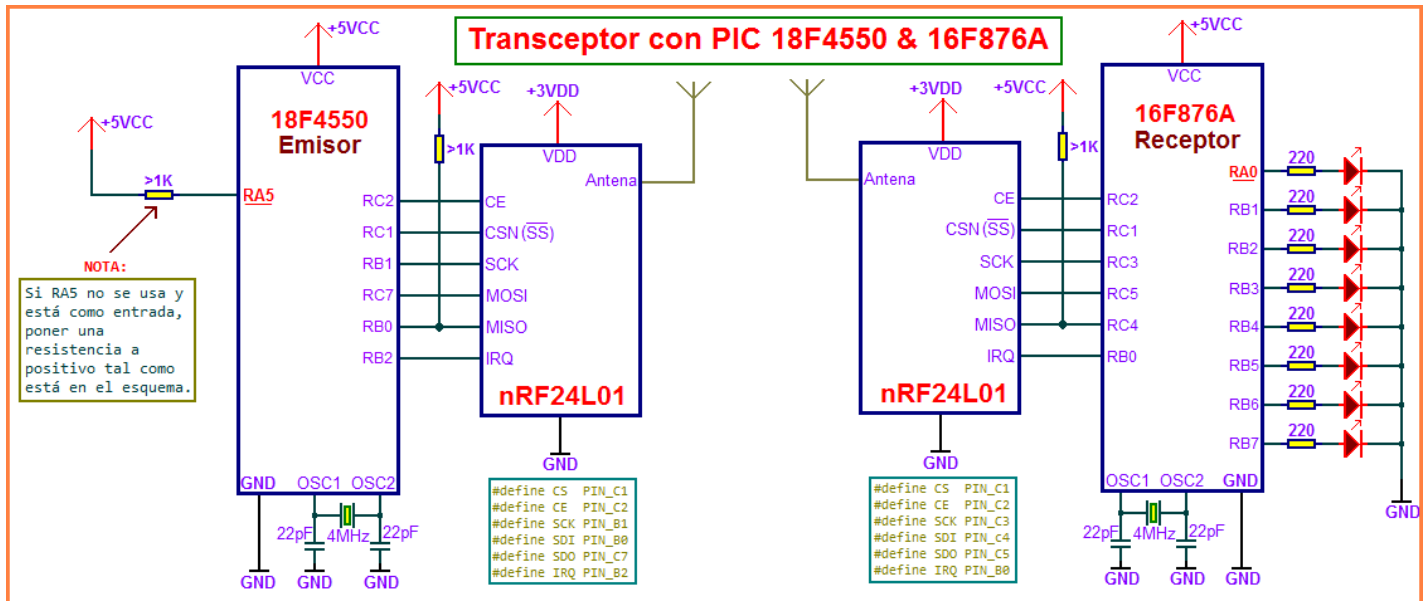




nRF24L01 & 18F4550

Este proyecto está realizado exclusivamente en CCS C (PIC C).



En este proyecto vamos a usar el PIC 18F4550 como emisor y el PIC 16F876 como receptor. Se trata de que el emisor envíe el conteo en binario de 0 a 255 al receptor. Me voy a centrar en la **modificación de la librería original "lib_rf2gh4_10.h"** para adaptarla al PIC 18F4550 (y a cualquier 18Fxx5x). Recuerda que cualquier cambio importante a nivel de hardware que hagas por tu cuenta luego has de tenerlo presente modificándolo también en la librería.

Si quieres saberlo todo sobre la librería, has de descargarte un estupendo manual que profundiza en ella para manejarla en CCS C (PIC C) y en ensamblador. Casi todas las cosas que explico en esta sección es gracias al manual que puedes descargar haciendo [clic aquí](#). El manual trata un transceptor llamado "BZI-RF2GH4" pero el chip que lleva ese transceptor es el nRF24L01, por tanto es totalmente compatible.

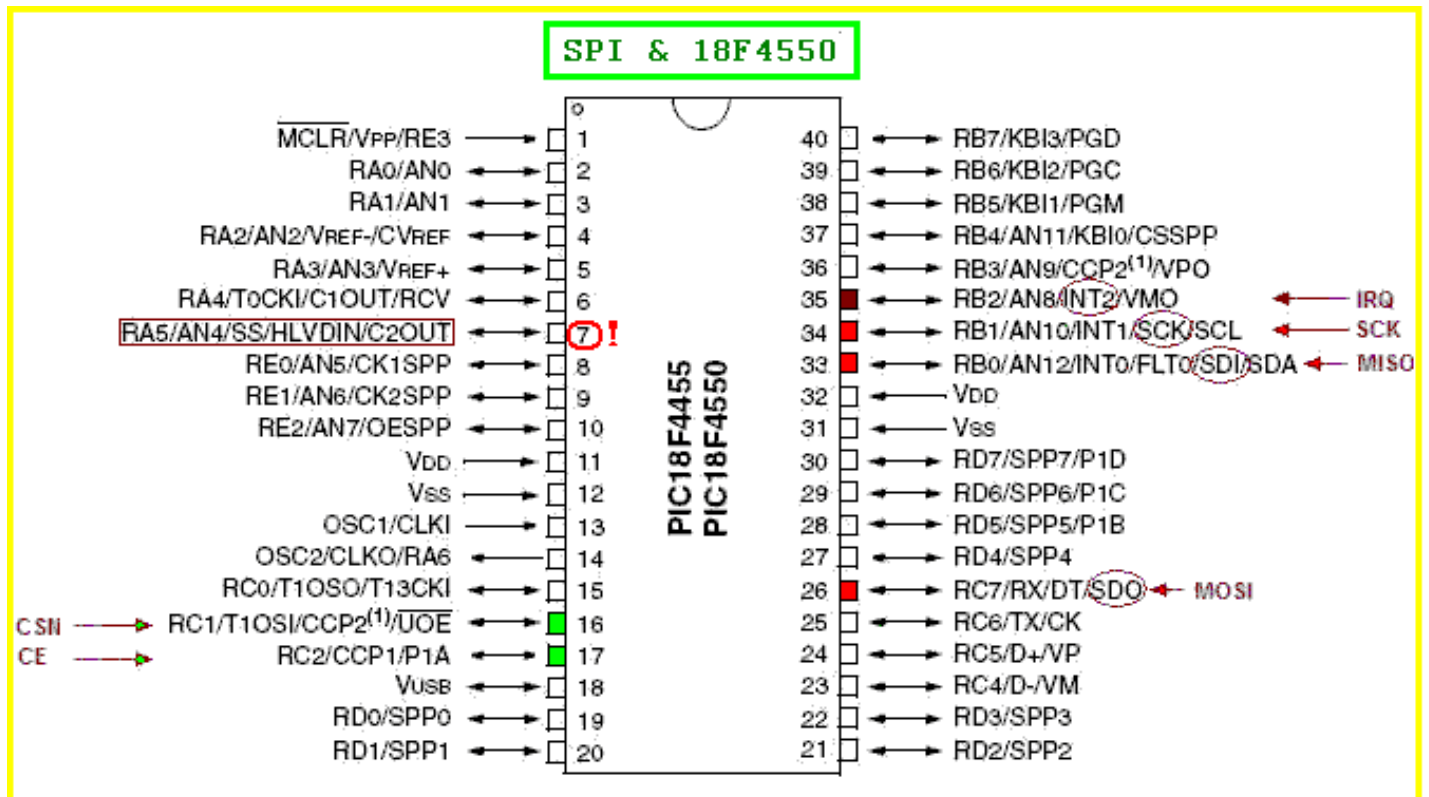
La librería la puedes descargar, junto a todo el proyecto, al final de esta página.

Como controlaremos el **SPI** del 18F4550 por hardware con el nRF24L01, recuerda que hay tres patillas que no se pueden modificar de lugar y son:



~~MISO~~ del nRF24L01 a ----> -RB0-
 MOSI del nRF24L01 a ----> -RC7-
 SCK del nRF24L01 a ----> -RB1-

Se han de corresponder con hardware del PIC que uses, en este caso me referiré al 18F4550.



Existen dos patillas del SPI que sí puedes modificar de lugar y son **CE** y **CSN**. Esas patillas del nRF24L01 las pongo en C2 y C1 respectivamente.

Ahora modificaremos la librería "[lib_rf2gh4_10.h](#)" que usamos en el [apartado anterior](#) para 16F876 y la adaptaremos al **18F4550**. Haremos cambios en las definiciones de pines I/O, TRIS y la configuración de RB2 como interrupción externa. En el ZIP de descarga del proyecto ya viene todo modificado, no hay que tocar nada por el momento. En esta sección explico las modificaciones que fui haciendo de la librería original.

Esto son los cambios:

```

#define RF_CS PIN_C1 // Definición de los pines del PIC para controlar
el nRF24L01.
#define RF_CE PIN_C2
#define SCK PIN_B1
#define SDI PIN_B0
#define SDO PIN_C7
#define RF_IRQ PIN_B2
#define RF_CS TRIS TRISC 1 // Definición de los pines TRIS del PIC
  
```



```

#define RF_CS_TRIS TRISC,1 // Definición de los pines TRIS del PIC.
#define RF_CE_TRIS TRISC,2
#define SCK_TRIS TRISB,1
#define SDI_TRIS TRISB,0
#define SDO_TRIS TRISC,7
#define RF_IRQ_TRIS TRISB,2
//*****
//* VARIABLES *
//*****
#BYTE TRISA = 0xF92 // Dirección de los TRIS A, B, C, D y E.
#BYTE TRISB = 0xF93
#BYTE TRISC = 0xF94
#BYTE TRISD = 0xF95
#BYTE TRISE = 0xF96
#BYTE INTCON = 0xFF2 // Dirección del registro de interrupciones.

```

En la parte de la librería que configura la interrupción externa, como usaremos RB0 y RB1 para el SPI, hemos de colocar dicha interrupción en otro lugar y tenemos la suerte de que el PIC 18F4550 tiene tres interrupciones externas (RB0, RB1 y RB2), así que podemos coger la última, es decir en RB2. La configuramos de la siguiente manera:

```

void RF_INT_EN()
{
    disable_interrupts(global);
    enable_interrupts(int_ext2); //Habilita interrupción externa RB2
    por flanco de bajada.
    ext_int_edge(2, H_TO_L );
    bit_set(RF_IRQ_TRIS);
    enable_interrupts(global);
}

```

Y referente a las interrupciones, casi al final de la librería hay tres "**clear_interrupt(int_ext);**" los he cambiado por "**clear_interrupt(int_ext2);**" para borrar la interrupción que corresponde a RB2 para los PICs 18Fxx5x.

Otro detalle que se puede tocar en la librería es la velocidad de transmisión, donde podemos configurar 1Mbps ó 2Mbps. Ya dije en la [introducción](#) de esta sección lo siguiente:

>>

La distancia que quieras alcanzar dependerá de la velocidad de transmisión. En el peor de los casos podrás comunicarte hasta 8 metros de distancia al aire libre (configurando a 2Mbps), y en el mejor de los casos hasta 40 metros (configurando a 1Mbps) con el nRF24L01 más barato (existen varios tipos). Has de tener presente que las micro-ondas (las mismas que usa los "wifis", es decir unos 2.4GHz) si encuentra una pared, un muro o un pasillo, la señal se debilita notablemente.

<<



En la librería hay una parte que configura la potencia de salida (en decibelios), la ganancia de entrada

(LNA) y la velocidad de transferencia. Por defecto está configurada todos los valores al máximo
 decir: máxima potencia de salida, máxima ganancia y máxima velocidad de transferencia.

Hay un registro llamado **RF_SETUP**, del cual sólo son útiles los bits 0 al 3, el resto no se usan; aunque el bit 4 es para hacer test, pero ese no nos interesa y seguirá quedando a cero.

El bit 0: Si lo pones a 1 activas el LNA, es decir, que tendrás la máxima ganancia (sensibilidad). Si lo pones a cero funcionará sin ganancia.

Los bits 1 y 2: Tendríamos 4 combinaciones que van del 00 (mínima potencia de salida y mínimo consumo) al 11 (máxima potencia de salida y máximo consumo).

El bit 3: Velocidad de transferencia. Puesto a 1 la transferencia es de 2Mbps, y a 0 sería de 1Mbps.

Dentro de la librería, buscando un poco, encontraremos esto:

```
// RF_SETUP
// Configuración aspectos RF.
// Ganancia máxima de LNA, 0dBm potencia de salida y 2Mbps de velocidad.
output_low(RF_CS);
spi_write(0x26);
spi_write(0x0F);
output_high(RF_CS);
```

Nos interesa la parte que pone "spi_write(0x0F);"

Es 0x0F porque los bits 0 al 3 están a 1, es decir: 1111, que mirando los bits antes explicados te configura por defecto todo al máximo. Así que lo único que hemos de hacer es cambiar la máxima transferencia por la mínima, es decir, pasar de 2Mbps a 1Mbps, así obtendremos la posibilidad de poder transmitir información hasta 40 metros en campo abierto. Eso significa poner el bit 3 a cero. Nos quedaría 0111, por tanto es 7. Así que sólo hemos de cambiar esa parte y quedaría así:

```
// RF_SETUP
// Configuración aspectos RF.
// Ganancia máxima de LNA, 0dBm potencia de salida y 1Mbps de velocidad.
output_low(RF_CS);
spi_write(0x26);
spi_write(0x07); // <----- Aquí!!!
output_high(RF_CS);
```

Hay que tener presente que la comunicación (SPI) entre el PIC y el transceptor no conviene que supere los 8MHz, lo cual quiere decir que un PIC con frecuencia de 32MHz ya estaría en el límite. Los 32MHz se debe a que $32(\text{MHz})/4(\text{ciclos de reloj por instrucción})=8\text{MHz}$ reales. En el ejemplo que pongo aquí funciona a 48MHz y sobrepasa ese límite sin embargo funciona bien y supongo que es debido a la configuración de la comunicación SPI y por ⁽ⁱ⁾ verlo en la librería como: (línea 63.)

```

X setup_spi(SPI_MASTER|SPI_L_TO_H|SPI_XMIT_L_TO_H|SPI_CLK_DIV_16|SPI_SAMP_
E_AT_END);

```

Originalmente viene como **SPI_CLK_DIV_4** y para que no hubiese problemas en este sentido lo cambié por **SPI_CLK_DIV_16**. Sea como fuere déjalo así y cuando todo te vaya bien pruebas a bajar el divisor del clock para el SPI.

Y para terminar sobre la modificación de la librería comento dos puntos "calientes" que hace referencia al acuse de recibo (ACK). Son las líneas 424 y 462 que hace referencia a un contador de 16 bits llamado "noRF". La velocidad de ese contador correrá más o menos rápido dependiendo de la frecuencia del cristal o del PLL interno del PIC que estés usando, sin embargo dicho contador sirve para poner un tiempo límite máximo de 7ms (corresponde a un cristal de 4MHz sin PLL) durante el cual espera el acuse de recibo (ACK). Como estoy usando el PLL interno para ponerlo a 48MHz, ese tiempo se acorta y es por ello que lo he aumentado a **6000** en lugar de 500. Si para 4MHz era 500, para 48MHz son 6000, así mantengo esos 7ms de espera para el ACK. En el ejemplo que pongo a continuación no uso el acuse de recibo (ACK) pero para otros proyectos quizás tengas que tenerlo presente.

Veamos el código en CCS para el PIC 18F4550, programa emisor:

```

#include <18F4550.h>
#fuses
XTPLL,PLL1,CPUDIV1,PUT,NOBROWNOUT,NOVREGEN,NOWDT,NOPBADEN,MCLR,NOLVP,NOD
EBUG,NOPROTECT,NOCPD,USBDIV
#use delay(clock=48000000)
#include "lib_rf2gh4_10_4550.h" // Librería modificada del nRF24L01 para
el manejo SPI con el PIC 18F4550.
#byte porta=0xF80 // Dirección de los puertos A, B, C, D y
E.
#byte portb=0xF81
#byte portc=0xF82
#byte portd=0xF83
#byte porte=0xF84
#int_ext2 // Esta rutina está para un futuro si haces
comunicaciones bidireccionales.
void int_RB2() // Si te da error esta línea, sustituir
por: void int_ext_isr(void).
{ // No tiene efecto en el programa
principal, ya que sólo emite.
int8 ret1; // Se encargaría de la recepción de datos.

ret1 = RF_RECEIVE();
if ( (ret1 == 0) || (ret1 == 1) )
{
do
{
ret1 = RF_RECEIVE(); i
} while ( (ret1 == 0) || (ret1 == 1) );
}

```

```

}
}
void main()
{
    RF_INT_EN();          // Habilitar interrupción RB2/INT2 en este
PIC.
    RF_CONFIG_SPI();      // Configurar módulo SPI del PIC.
    RF_CONFIG(0x40,0x01); // Configurar módulo RF canal y dirección
de recepción de datos para este PIC.
    RF_ON();              // Activar el módulo RF.

    delay_ms(5);          // Dejamos como mínimo 2.5ms antes de
comenzar a enviar.

    set_tris_a(0b011111); // Dejo RA5 como salida para evitar poner
la resistencia de polarización.

    int8 cont=0, ret2;    // Declaramos las variables a utilizar.

    while(true)          // Bucle infinito.
    {
        RF_DATA[0] = cont; // El contenido del contador lo cargo en
RF_DATA[0] para ser enviado.
        RF_DIR=0x08;      // Dirección de envío para el receptor.
        ret2=RF_SEND();   // Envía el dato.
        cont++;           // Incrementa el contador.
        delay_ms(50);     // Una pausa en cada incremento.
    }
}

```

* La configuración de los #fuses está para un cristal de 4MHz. Un PLL interno consigue que el PIC funcione a 48MHz. Para otras configuraciones del cristal y que funcione a 48MHz, haz [clic aquí](#).

* La interrupción **#int_ext2** en realidad no llega a utilizarse, la he puesto para futuros proyectos en el que quieras hacer recepción de datos además de emitir. Si le llegara algún dato, tal y como está en el programa, lo único que haría sería vaciar el buffer de recepción, nada más; no afecta al programa principal. Pese a lo dicho, no elimines esta parte del programa principal. Luego, cuando todo te funcione bien, haz los cambios que quieras.

* Observa la parte que pone: **#include "lib_rf2gh4_10_4550.h"** A la librería le puse otro nombre para advertir que está modificada para el PIC 18F4550.

* **RF_INT_EN();** Ese comando habilita la recepción de datos por interrupción de la RB2/INT2, pero como sólo vamos a emitir, está puesta simbólicamente para futuros proyectos en el que sí haya recepción de datos. De todas formas no elimines esta parte.

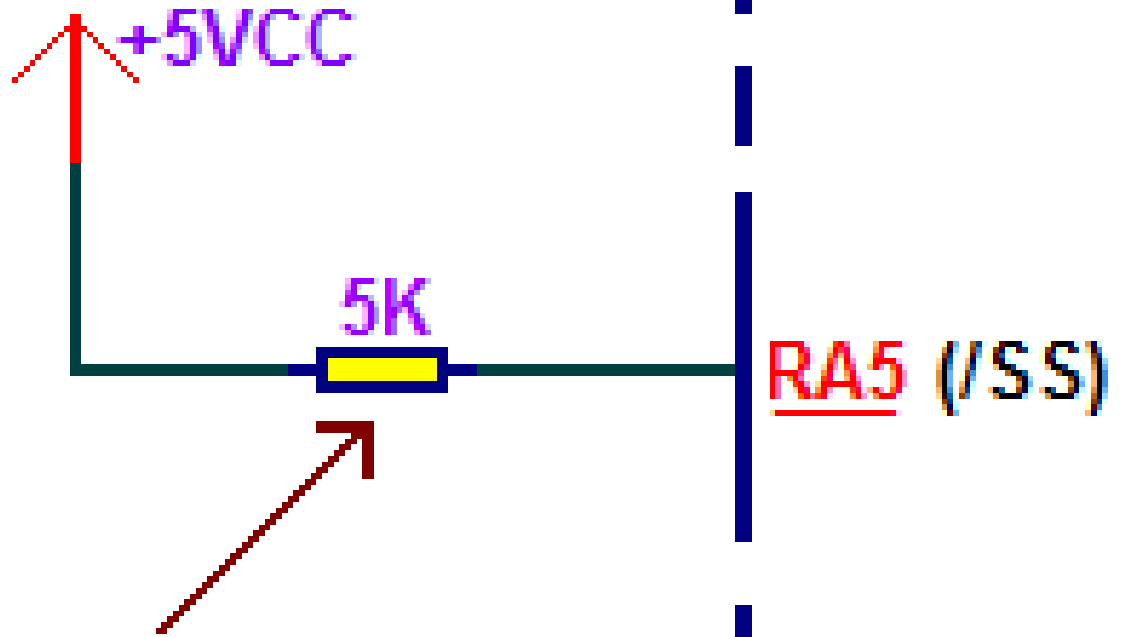
* **RF_CONFIG_SPI();** Configura el SPI entre el PIC y el nRF24L01, definiendo los pines, TRIS e interrupción si la hubiera. Estas definiciones ⁽ⁱ⁾ están dentro de la librería **"lib_rf2gh4_10_4550.h"**.

* **RF_CONFIG(0x40,0x01);** Se refiere "**0x40**" al canal de comunicación, este siempre ha de ser el mismo para todos los PICs que se quieran comunicar entre sí. "**0x01**" es la dirección de recepción. Aunque este ejercicio no va a recibir datos, de todas formas hay que poner una dirección de recepción.

* **RF_ON();** Activa el módulo RF. Después de este comando conviene que transcurra unos 2.5 milisegundos antes de hacer envíos o recepciones de datos.

* **set_tris_a(0b011111);** Todo el puerto **A** lo convertimos en entradas de datos a excepción de RA5 que lo ponemos como salida. Ese bit se podría haber puesto como entrada porque no se usa, pero lo he dejado como salida por si se te olvida poner en esa patilla una resistencia a positivo. Si haces cualquier otro proyecto diferente a este has de saber que **RA5 (/SS)** del PIC 18F4550, si está como entrada y al aire ha de polarizarse esa entrada a positivo mediante una resistencia, como en la imagen de abajo. No debe de quedar al aire.



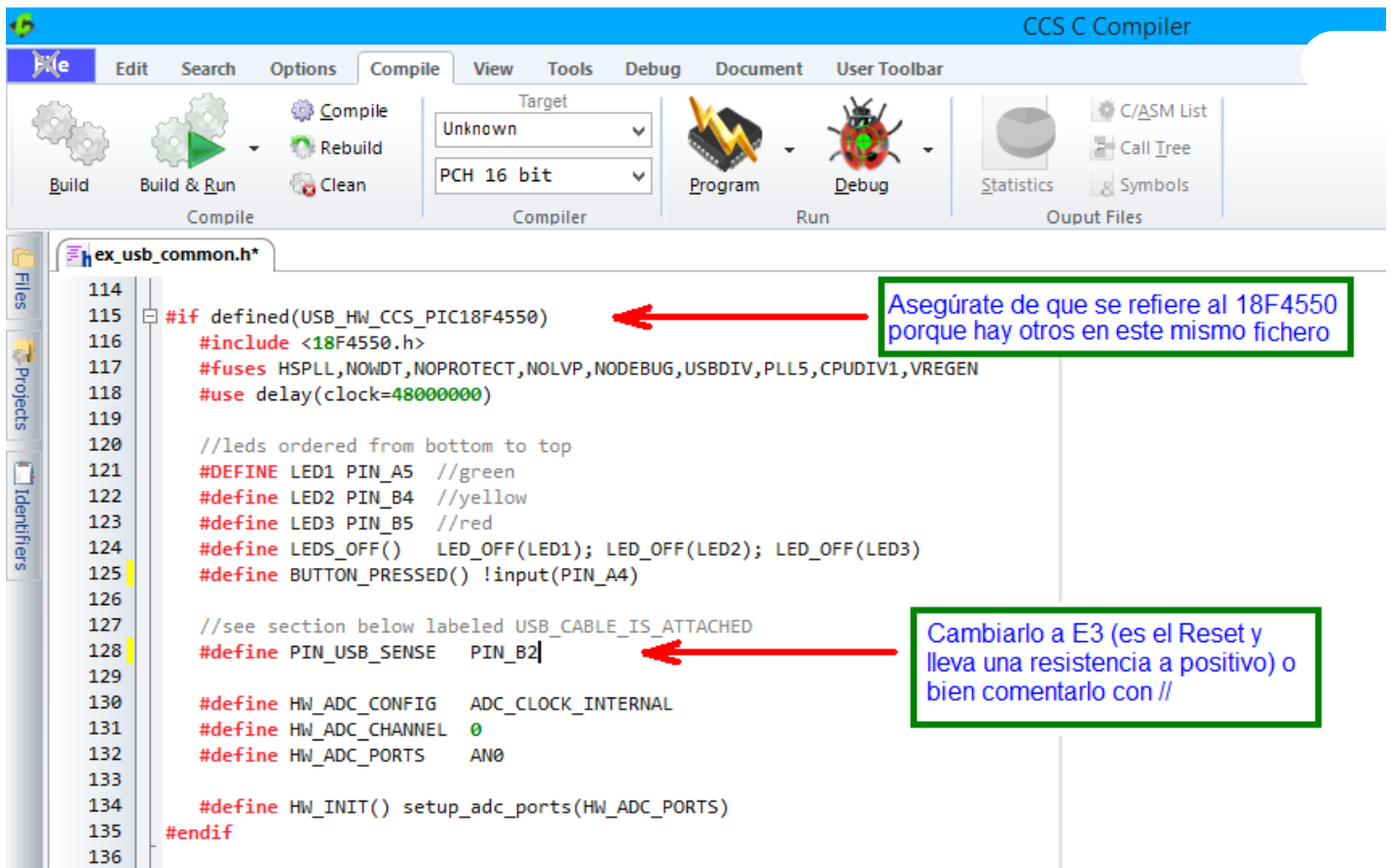


NOTA:

Si RA5 no se usa y está como entrada, poner una resistencia a positivo tal como está en el esquema.

Lo ideal sería poner resistencias a positivo a todas las patillas que no se usan para evitar que se cuele ruido estático. La patilla B2 es otro lugar por donde se puede colar ruido. En esta imagen puedes ver el fichero "ex_usb_common.h" y cómo modificar esa parte. Aunque no utilices el USB, si estás usando el bootloader a través de USB para el PIC18F4550 sería recomendable modificar la parte que se indica.





Si las patillas de los PICs no está conectada a nada, es decir al aire, como son circuitos CMOS son sensibles a la electricidad estática del ambiente en algunos casos. Por ejemplo, al pasar la mano cerca del PIC podría colarse ruido y el PIC adquirir un extraño comportamiento. Esto puede suceder especialmente en el 18F4550. Así que si fuera posible toda patilla que no se esté utilizando o no esté conectada físicamente a algo deberían de quedar polarizadas a positivo a través de resistencias.

* **RF_DATA[0]=cont;** Carga el contenido de la variable "cont" en RF_DATA[0]. Tienes 8 bytes, de RF_DATA[0] a RF_DATA[7]. Si quieres aprovechar todo el ancho de banda que da el nRF24L01 tendrías que usar los 8 bytes disponibles para enviar (y recibir) información. También lo puedes ver como la posibilidad de enviar 8 bytes de una vez aunque aquí sólo usamos uno porque no necesitamos más.

* **RF_DIR=0x08;** Este comando le está diciendo a qué dirección ha de ir la información que quieras enviar. Piensa que puedes tener varios PICs de recepción (127 teóricos y 32 prácticos), pues irá al PIC que tenga configurado como recepción la dirección 0x08.

* **ret2=RF_SEND();** Envía la información. "ret2" nos sirve para saber qué ha sucedido al intentar enviar. Se cargará con '0' si se ha enviado y ha recibido ACK (confirmación de recepción); '1' si ha enviado pero no ha recibido el ACK; y '2' si no ha podido ser enviado (fallo del hardware del emisor). Se podría poner "**RF_SEND();**" a secas para enviar, pero ortodoxamente lo correcto es hacerlo tal como lo he puesto.

Pasamos a ver el código del receptor. Recuerda que ahora usaremos el PIC 16F876:

```

#include <16F876A.h>
#FUSES NOWDT, XT, PUT, NOPROT ⓘ, NODEBUG, NOBROWNOUT, NOLVP, NOCPD,
NOWRT

```

```

#include <del>lib_rf2gh4_10.h</del> // Librería para manejar el módulo SPI con
nRF24L01.
#define porta=0x05 // Dirección del puerto A.
#define portb=0x06 // Dirección del puerto B.
#define a0=porta.0 // Un truco para más tarde hacer intercambios
de bits.
int8 ret1, data;
#define b0=data.0 // Un truco para más tarde hacer intercambios
de bits.
#define int_ext // Interrupción RB0/INT para el módulo RF.
void int_RB0() // Si te da error esta línea, sustituir por:
void int_ext_isr(void).
{
    ret1 = RF_RECEIVE();
    if ( (ret1 == 0) || (ret1 == 1) )
    {
        do
        {
            data=RF_DATA[0]; // Data contendrá el valor que le llegue del
emisor, a través de RF_DATA[0].
            portb=data; // Lo que haya en data lo refleja en los
LEDs.
            a0=b0; // Un truco para que RB0 sirva de
interrupción y RA0 para poner el LED correspondiente.
            ret1 = RF_RECEIVE();
        } while ( (ret1 == 0) || (ret1 == 1) ); // Tanto si existe entrada
múltiple o simple de datos los lee.
    }
}
void main() //Programa principal.
{
    set_tris_a(0b111110); // RA0 sustituye a RB0 y se pone un LED en
RA0.
    set_tris_b(0b00000001); // RB0 es para la interrupción, el resto
son LEDs.
    portb=0;

    RF_INT_EN(); // Habilitar interrupción RB0/INT.
    RF_CONFIG_SPI(); // Configurar módulos SPI del PIC.
    RF_CONFIG(0x40,0x08); // Configurar módulo RF (canal y
dirección).
    RF_ON(); // Activar el módulo RF.

    while(true); // Bucle infinito.
}

```

* La interrupción **#int_ext** se encargará de recibir los datos nada más llegar.

* No debemos de confundir "ret2" (del emisor) con "ret1" (del receptor), porque en esas variables se guardará también valores "0, 1, 2...". Tienen significados distintos para emitir que para recibir. En el receptor, cuando vemos esto: **ret1 = RF_RECEIVE();** "ret1" nos estará diciendo que, si el valor es '0' hay entrada simple de datos (una sola recepción, o dicho de otra manera, los bytes RF_DATA[0] a RF_DATA[7] están listos para ser leídos); si nos da '1' significa que hay más de una entrada (cada entrada es de 8 bytes) para ser leída, es decir, una segunda o tercera tanda, y no soporta más de tres niveles, el resto se perderían. Cuando "ret1" nos dé el valor '2', significa que no hay entradas para leer, o sea, no ha habido entrada nueva de datos. En el programa, tanto si hay entrada simple como múltiple, me quedo con lo último que ha llegado. El buffer (compuesto de 8 bytes por 3 niveles) queda vacío una vez que sale del "**while ((ret1 == 0) || (ret1 == 1));**". Hay que procurar que el emisor sea un poco más lento que el receptor para evitar saturar al receptor y eso puede suceder cuando ret1 nos devuelve el valor '1' (recepción múltiple).

* **RF_INT_EN();** Este comando habilita la recepción de datos por interrupción de la RB0/INT. Una vez que llegan los datos los puedes tratar inmediatamente, en la misma interrupción, en el ejemplo propuesto lo hacemos de esta forma.


* **RF_CONFIG_SPI();** Configura el SPI entre el PIC y el nRF24L01, definiendo los patillajes, TRIS e interrupción si la hubiera. Estas definiciones están en la librería "**lib_rf2gh4_10.h**"

* **RF_CONFIG(0x40,0x08);** Se refiere "**0x40**" al canal de comunicación, este siempre ha de ser el mismo para todos los PICs que quieras comunicar entre sí. "**0x08**" es la dirección de recepción. Recuerda que en el emisor pusimos: **RF_DIR=0x08;** como dirección de envío, así que si el transceptor receptor está configurado con esa dirección podrá recibir esos datos, si no fuese esa dirección el transceptor entenderá que se refiere a otro y no a él, por tanto no atendería a esos datos.

* **RF_ON();** Activa el módulo RF. Después de este comando conviene que transcurra unos 2.5 milisegundos antes de hacer envíos o recepciones de datos.

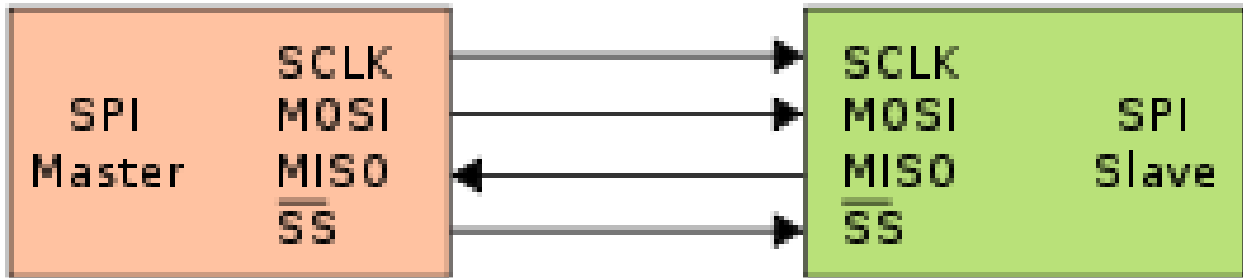
* **set_tris_b(0b00000001);** Casi todo el puerto B lo convertimos en salidas de datos, exceptuando RB0 que se usa para la interrupción. Observa en el esquema sustituyo RB0 por **RA0** en el PIC receptor. Utilizo un truco de intercambios de bits, a través de las dos directivas **#bit** que hay en el programa, en el que RA0 tomará el valor que hubiese ido a RB0.

* **do {...} while((ret1 == 0) || (ret1 == 1));** Mientras haya datos por leer eso estará haciendo. Pero aquí hay que entender bien una cosa: el nRF24L01 envía y recibe 8 bytes de un vez (es un array de 8 bytes y nosotros sólo usamos el primero), esto significa que si hay más datos por leer, leerá otros 8 bytes y hay que cargar esos datos en variables o un array que hayamos declarado nosotros antes de hacer **ret1 = RF_RECEIVE();** Esto sucede cuando hay entrada múltiple de datos y sólo soporta 3 niveles, si hubiese más de tres tandas, el resto se perderían. Es por ello que para un funcionamiento simple y sin problemas se recomienda que el emisor emita un poco más lento de lo que el receptor puede asimilar. En el ejemplo que pongo aquí lo que hace es quedarse con lo último que haya recibido.

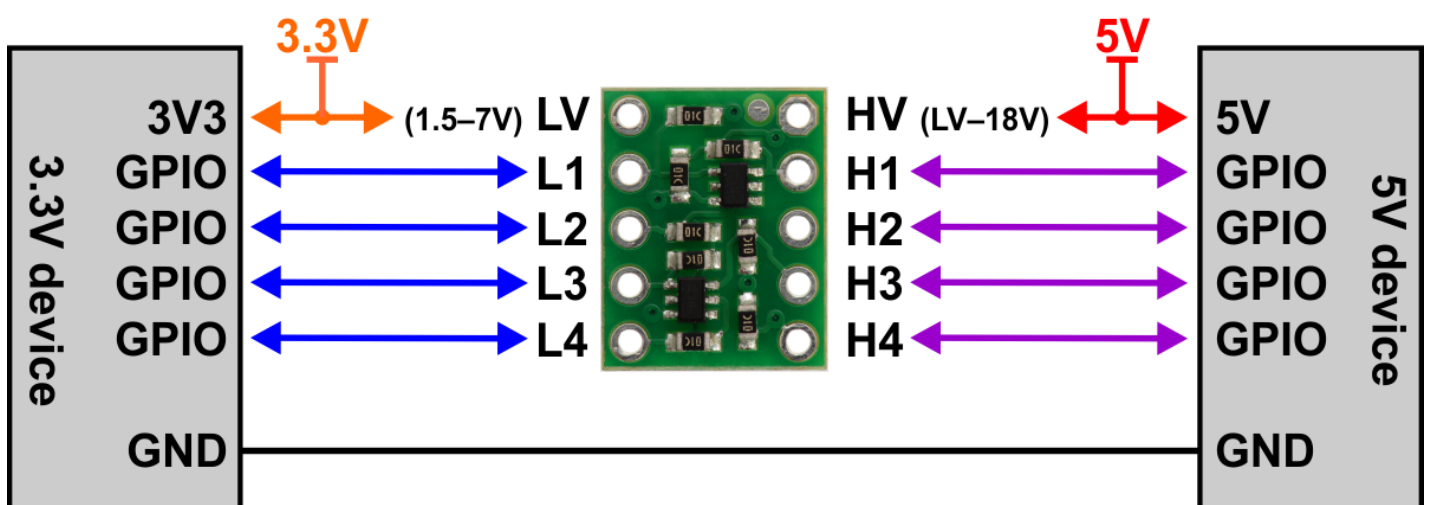
* En la recepción de datos, si esos datos  son tratados inmediatamente los puedes perder. Es decir, que una vez que le lleguen datos al receptor has de cargarlos en una variable o matriz (según el caso)

que una vez que le lleguen datos al receptor nos de cargarlos en una variable o matriz (según el caso) antes de que pase al programa principal. En este caso la variable "data" me sirve para poder tratar BIT 0 de esa variable para extraerlo por RA0.

Un detalle sobre el hardware.



En nuestro proyecto tenemos siempre a los PICs como maestros [SPI](#) y los transceptores como esclavos. Dado que los PICs que estamos usando funcionan con 5 voltios y los nRF24L01 funcionan con un rango de 3.3 voltios puede haber una descompensación de voltajes, en el sentido de: ¿cuándo considerar que hay un 1 lógico y cuándo un 0 en una entrada del PIC? Esto puede suceder en la única entrada SPI que tiene el PIC, que es la patilla **MISO**. Es por ello que se ha de poner una resistencia a positivo de 5 voltios en dicha patilla (tal como está en el esquema, al comienzo de esta página), para compensar la tensión y cuando le llegue un 1 lógico al PIC lo lea como real. Tener este simple gesto nos ahorrará muchos quebraderos de cabeza. Lo ideal sería utilizar [convertidores de nivel](#), son bidireccionales, es decir, no importa si es una entrada o salida de información, y a día de hoy están realmente baratos. Un esquema-ejemplo cualquiera sería la siguiente imagen:



Está claro que si adaptas los programas a otros PICs que sí pueden funcionar a 3.3V o tolera tensiones (como por ejemplo Arduino) no te hará falta nada de esto y podrás conectarlo directamente, sin resistencia de polarización o conversores de tensión.

Un consejo: Muchas personas han tenido problemas para hacer funcionar otros proyectos parecidos y es que estos transceptores son bastante puñeteros. Mi consejo es realizar el proyecto tal como está aquí, usando el esquema, los programas y librería que está en el zip de descarga. Una vez te funcione todo bien, puedes comenzar a realizar los cambios necesarios hasta convertirlo en tu propio proyecto. Si eres amante del paracetamol y el ibuprofeno entonces hazlo todo según tu criterio desde el comienzo.

Para saber más sobre las funciones `RF_RECEIVE()` y `RF_SEND()` haz [clic aquí](#).

[Descargar todo el proyecto](#)

Contiene código fuente, librería, HEX y esquema.

