

CONEXION SERIAL CON EL PIC rev2 *

Paul Aguayo S., paguayo@olimex.cl

7 de septiembre de 2005

*Basado tutorial de sparkfun

Índice

1. Requisitos	3
2. ¿Que hace exactamente el chip MAX232?	3
3. Configurando el UART	3
4. Transmisión de Datos	3
5. El cable	4
6. Empezando a transmitir	4
7. Debugging	4
8. Podemos transmitir pero no recibir	4
9. stdio.c	6
10. 232_comm.c	14

1. Requisitos

- Tarjeta de desarrollo de 18 Pines + PIC16F628
- Editor de Texto JFE
- Compilador CC5X
- 232-comm.c
- stdio.c

2. ¿Que hace exactamente el chip MAX232?

El MAX232 es el circuito integrado estandar para convertir señales TTL/CMOS a señales RS232. Las señales en RS232 tienen 1's y 0's estos son +12V y -12V respectivamente. Las señales de salida del PIC son entre 0 y 5V.

Lo que el MAX232 hace es poner 12V en el pin T1OUT cuando se alimenta con 5V el pin T1IN. De esta forma se pueden pasar datos hacia tu computador. Si presionas una tecla en el hyperterminal una señal será enviada a través del cable hasta el R1IN del MAX232 donde la señal de 12V proveniente del PC es convertida a 0-5V. Esta señal que sale por el R1OUT es perfectamente manejable por el PIC. Todo OK. La magia está en como el MAX232, el cual es alimentado con 5V y puede producir señales de +/-12V. Bueno, esto se hace con condensadores externos.

Muchas de las tarjetas de Olimex vienen equipadas con el MAX232 y un conector DB9. Esto es muy útil, pero **debes conectar** en MAX232 al PIC a mano.

Nota: La impresión en la tarjeta puede ser mal entendida. El pad 'TX' es el pad que debe ser conectado en el 'RX' de tu microcontrolador. ¿Por qué lo hicieron así?. La respuesta es por que depende del lado del que se mire. El MAX232 transmite al puerto receptor del PIC. El MAX232 recibe datos desde el puerto TX del PIC. Espero que no sea tan confuso.

Saca tu viejo caufín y soldadura, encuentra el TX del MAX232, debes soldar este pad con el pin7 del 16F628. También debes soldar el RX al pin 8 del 16F628. No te preocupes del CTS y el RTS.

Tu placa ahora esta lista para transmitir en RS232. Las señales debieran ser fuertes y claras. Ahora hay que configurar el PIC para que la salida sea a 9600 baudios.

3. Configurando el UART

Muchos PIC's tienen el módulo UART (Universale Asynchronous Receive/Transmit). Revisa el datasheet de tu PIC. Los beneficios de utilizar el UART en vez de algún tipo de control por software es que una vez que éste esté configurado correctamente, sólo tienes que poner un byte de datos en el buffer de salida y el hardware se encarga del resto.

Lee el datasheet

Sólo podemos llevarte de la mano por poco tiempo, tarde o temprano vas a tener que leer el datasheet y ver como funciona el UART.

4. Transmisión de Datos

Mira el archivo stdio.c. Primero discutiremos la función enable_uart_TX(uns8).

La primera configuración que hay que hacer es la velocidad de transmisión. Esto se hace en el registro SPBRG. Este registro se utiliza como timing para la comunicación, en nuestro caso (@20MHz) SPBRG=32 listo!

Ahora pongamos el BRGH = 0 (Baud Rate Generator High Speed).

SYNC tiene que estar en 0, SYNC=0. El puerto serial esta listo para ser utilizado: SPEN = 1

Probablemente no quieras habilitar las interrupciones. Por ahora no nos preocupemos del TXIE - Transmit Interrupt Enable. Las interrupciones son un dolor de cabeza, asegúrate de tener todo el resto funcionando antes de empezar a jugar con las interrupciones.

La última línea del código habilita la transmisión. Si algo es escrito en el TXREG (transmission register) será puesto en el buffer de transmisión y enviado a través del cable.

Tan pronto como la transmisión esté completa, el TXIF - transmisión interrupt flag - levantará una interrupción. Pero en nuestro caso NO!

En este punto ya debieramos tener nuestro PIC configurado.

5. El cable

Para que la comunicación se lleve a cabo es necesario tener un conector de 9 pines (DB9, cable serial) puesto en el conector hembra de la placa y por el otro lado enchufado en el puerto serial de tu PC. Ahora si que estamos listos.

6. Empezando a transmitir

Como se describió anteriormente, cuando se envía algo al registro TXREG, éste se va inmediatamente. Entonces todo lo que necesitamos es esperar que el TXIF este activado. Cuando esto sucede la transmisión está terminada y podemos enviar el siguiente byte.

Miremos ahora la función `rs_out(uns8)`. Pasamos el byte que se desea enviar a la función `rs_out`. `rs_out` toma este byte y lo envía al registro TXREG, y luego espera que el TXIF este activo (espera que la transmisión termine). El UART obedientemente envía el byte bit a bit al MAX232. El MAX232 traduce los bits individuales y los envía por la línea hacia el puerto serial del computador. Si todo sale bien entonces el caracter ASCII equivalente al byte enviado aparecerá en la pantalla del hyperterminal.

Esto es todo lo que hay que hacer para enviar un caracter ASCII al computador.

Nota sobre el Hyperterminal: Todas las comunicaciones descritas en este tutorial son a 9600bps, con 8 bits, sin bit de paridad y sin start/stop bit. Asegúrate de configurar el hyperterminal de esta forma 9600 8-N-1.

7. Debugging

La mejor herramienta de debug es el famoso `printf`. Mira como está hecha esta función en el archivo `stdio.c`. Esta función toma un string y lo envía por el puerto serial byte a byte.

Ejemplo: `printf("Hola Mundo!",0);`

Este en ejemplo se pasa en string 'Hola Mundo!' a la función `printf`. Después cada letra es enviada a la función `rs_out` y de esa forma se obtiene la palabra completa.

Ahora algo complicado:

Ejemplo: `printf("Contador : %h\n\r", mi_contador);`

En este ejemplo imprimirá el string 'Contador :', pero cuando el `printf` llega al `%h`, este tomara el valor de `mi_contador` y lo imprimirá como su valor hex ("0xAC" por ejemplo) en la pantalla del hyperterminal. También imprimirá el salto de línea y el retorno.

Antes de que empieces a escribir unas cuantas líneas de código una advertencia: las rutinas para transmisión como el `printf(rs_out, rs_out_bb, printf_bb, etc)` son muy consumidoras de procesador y agrega varias líneas a tu código. El archivo `stdio.c` es muy útil para debugging pero desperdicia muchos ciclos innecesariamente.

8. Podemos transmitir pero no recibir

Sería excelente si pudiéramos enviar comandos al PIC desde el teclado. Es decir, algo como presionar la tecla 'k' y obtener un 'Hola', presionar la letra 'w' y obtener un 'chao' en la pantalla del hyperterminal.

Para hacer esto será necesario configurar la sección del UART encargada de la recepción. Es similar a configurar la parte de transmisión de la UART

Mira el código de la función `enable_uart_RX(bit)` en el archivo `stdio.c`

El SPBRG (baud rate generator) está configurado en 9600 baudios a 20MHz: `SPBRG = 32`. El generador hay que dejarlo a velocidad normal: `BRGH=0`

Habilitar el puerto serial SPEN = 1 y habilitar la recepción CREN = 1. El buffer de recepción inmediatamente comienza a monitorear el pin RX para la recepción de datos.

Nota: El buffer TRIS debe ser configurado de acuerdo a los pines RX y TX del PIC. El pin RX debe tener el correspondiente bit configurado en el TRIS; de esta forma se utiliza como entrada. El pin TX debe tener el correspondiente bit en el TRIS en 0; de esta forma se convierte en una salida.

Para el PIC 16F628, RB1 es el pin RX y RB2 es el pin TX. Para que el UART funcione correctamente hay que configurar estos pines de la siguiente forma:

```
TRISB = 0b.0000.0010; //0 - Output, 1 - Input - RB1 is RX for UART
```

El comando TRISB configurará todos los puertos como salida con excepción del RB1 el cual se configurará como entrada.

Quizás desees monitorear la variable RCREG para ver si han llegado datos. Esto funcionaría, pero como los datos de la entrada no se pueden predecir y no queremos perder ningún dato, solo las interrupciones pueden resolver este problema. Así que es necesario activar las interrupciones. Esto se hace activando el registro RXIE (receive interrupt enable) el PEIE (peripheral interrupt enable) y el GIE (global interrupt enable). El uso de estos bits está bien explicado en el datasheet. Con estos bits activados, tan pronto se llene el buffer de recepción el RXIF activará una interrupción, y la rutina de interrupción asociada comenzará a ejecutarse.

Veamos ahora la rutina de interrupción:

```
interrupt serverX(void)
{

int_save_registers
char sv_FSR = FSR; // save FSR if required

if(RCIF) //UART Recieve Interrupt
{

data_in = RCREG;
print_it = TRUE;

//No clearing RCIF, must clear RCREG
RCREG = 0;
}

FSR = sv_FSR; // restore FSR if saved
int_restore_registers

}
```

Esto puede parecer un poco complejo al principio, pero no es tan complicado y es muy poderoso. Si has leído algo sobre las interrupciones en los PICs vas a descubrir rápidamente que tan complejo puede llegar a ser. CC5X se llevará la mayoría de los dolores de cabeza por ti. El comando int_save_registers y el sv_FSR almacenarán el estado actual de los registros del PIC. Antes de que el PIC vuelva a hacer lo que estaba haciendo antes de entrar a la rutina de interrupción, el estado del PIC es restaurado. Para entender mejor este proceso sería bueno revisar la documentación de CC5X. Hay un archivo de ejemplo que viene en el zip de CC5X llamado Int16xx.c, es bueno mirarlo. La sección 6.3 del archivo CC5X.txt es muy útil también.

Para saber que fue lo que causó la interrupción se ocupan los flags (banderas). Cada interrupción tiene su propio flag, es sólo una cosa sentencias "IF" para saber desde donde se hizo la interrupción. Para nuestro ejemplo, el RCIF debiera ser la única interrupción habilitada. Si no se limpia apropiadamente la interrupción después de la rutina de interrupción, el PIC puede quedarse pegado en un loop infinito.

Ahora de acuerdo a nuestra rutina de interrupción si recibimos un byte en el pin RX, entonces el "print_it" será TRUE y guardaremos la información en la variable "data_in".

Veamos ahora la rutina principal. Mientras técnicamente estamos esperando que el `print_it` cambie, podemos agregar más código en la rutina principal sin preocuparnos de que se nos pierdan datos.

Asumiendo que `print_it = TRUE`, la función principal imprimirá los valores HEX y ASCII del byte entrante, que todo esta correctamente cableado, que tienes el PIC programado y la tarjeta está enchufada, entonces el PIC comenzará a monitorear la información RS232 entrante. Si presionas la h en el hyperterminal obtendrás una salida como:

```
Ascii: 0x68
```

```
Key: h
```

Ahora si te pones a teclear un rato obtendrás una pantalla llena de respuestas. Ahora presiona 'Z' (mayúscula) Whoa! La pantalla cambiará a un montón de caracteres. Cada vez que el PIC ve un dato de entrada, el dato es almacenado en un arreglo llamado 'memory_array'. Entonces cuando se presiona la 'Z' los 64 últimos caracteres presionados son escritos en la pantalla del hyperterminal.

Este ejemplo no es muy práctico, pero es bastante ilustrativo para aprender a manejar el puerto serial y las interrupciones.

9. `stdio.c`

```
/*
 5/21/02
 Nathan Seidle
 nathan.seidle@colorado.edu

 Serial Out Started on 5-21
 rs_out Perfected on 5-24

 1Wire Serial Comm works with 4MHz Xtal
 Connect Serial_Out to Pin2 on DB9 Serial Connector
 Connect Pin5 on DB9 Connector to Signal Ground
 9600 Baud 8-N-1

 5-21 My first real C and Pic program.
 5-24 Attempting 20MHz implementation
 5-25 20MHz works
 5-25 Serial In works at 4MHz
 5-25 Passing Strings 9:20
 5-25 Option Selection 9:45

 6-9 'Stdio.c' created. Printf working with %d and %h
 7-20 Added a longer delay after rs_out
      Trying to get 20MHz on the 16F873 - I think the XTal is bad.
      20MHz also needs 5V Vdd. Something I dont have.
 2-9-03 Overhauled the 4MHz timing. Serial out works very well now.

 6-16-03 Discovered how to pass string in cc5x
      void test(const char *str);
      test("zbcdefghij"); TXREG = str[1];

      Moved to hardware UART. Old STDIO will be in goodworks.

      Works great! Even got the special print characters (\n, \r, \0) to work.

*/
```

```

//Clock is defined in Delay.c!!!

int counter;

//Setup the hardware UART TX module
void enable_uart_TX(bit want_ints)
{

#ifdef Crazy_Osc
    #ifdef Baud_9600
        SPBRG = 32; //20MHz for 9600 Baud
    #endif
#endif

#ifdef HS_Osc
    #ifdef Baud_9600
        SPBRG = 32; //20MHz for 9600 Baud
    #endif

    #ifdef Baud_4800
        SPBRG = 64; //20MHz for 4800 Baud
    #endif
#endif

    BRGH = 0; //Normal speed UART

    SYNC = 0;
    SPEN = 1;

    if(want_ints) //Check if we want to turn on interrupts
    {
        TXIE = 1;
        PEIE = 1;
        GIE = 1;
    }

    TXEN = 1; //Enable transmission
}

//Setup the hardware UART RX module
void enable_uart_RX(bit want_ints)
{

#ifdef HS_Osc
    #ifdef Baud_9600
        SPBRG = 32; //20MHz for 9600 Baud
    #endif

    #ifdef Baud_4800
        SPBRG = 64; //20MHz for 4800 Baud
    #endif
#endif
}

```

```

BRGH = 0; //Normal speed UART

SYNC = 0;
SPEN = 1;

CREN = 1;

//WREN = 1;

if(want_ints) //Check if we want to turn on interrupts
{
    RCIE = 1;
    PEIE = 1;
    GIE = 1;
}

}

//Sends nate to the Transmit Register
void rs_out(uns8 nate)
{
    //TXEN = 1; //Enable transmission
    TXREG = nate;
    while(TXIF == 0);

    //TXREG = 0;
    //TXEN = 0; //Disable transmission
}

//Returns ASCII Decimal and Hex values
uns8 bin2Hex(char x)
{
    skip(x);
    #pragma return[16] = "0123456789ABCDEF"
}

//Prints a string including variables
void printf(const char *nate, uns8 my_byte)
{
    //#pragma rambank 1
    uns8 i, k, m;
    uns8 high_byte = 0, low_byte = 0;
    uns8 y, z;

    for(i = 0 ; ; i++)
    {
        k = nate[i];

        if (k == '\0')
            break;

        else if (k == '%') //Print var
        {

```

```

i++;
k = nate[i];

if (k == '\0')
    break;
else if (k == '\\') //Print special characters
{
    i++;
    k = nate[i];

    rs_out(k);

} //End Special Characters
else if (k == 'b') //Print Binary
{
    for( m = 0 ; m < 8 ; m++ )
    {
        if (my_byte.7 == 1) rs_out('1');
        if (my_byte.7 == 0) rs_out('0');
        if (m == 3) rs_out(' ');

        my_byte = my_byte << 1;
    }
} //End Binary
else if (k == 'd') //Print Decimal
{
    /*if (my_byte.7 == 1)
    {
        rs_out('-');
        my_byte.7 = 0;
    }*/

    z = my_byte;

    y = z / 100;
    z = z % 100;
    if (y != 0)
        rs_out(bin2Hex(y)); //Print 100s

    y = z / 10;
    z = z % 10;
    if (y != 0)
        rs_out(bin2Hex(y)); //Print 10s

    rs_out(bin2Hex(z)); //Print 1s
} //End Decimal
else if (k == 'h') //Print Hex
{
    high_byte.3 = my_byte.7;
    high_byte.2 = my_byte.6;
    high_byte.1 = my_byte.5;
    high_byte.0 = my_byte.4;

    low_byte.3 = my_byte.3;

```

```

        low_byte.2 = my_byte.2;
        low_byte.1 = my_byte.1;
        low_byte.0 = my_byte.0;

        rs_out('0');
        rs_out('x');

        rs_out(bin2Hex(high_byte));
        rs_out(bin2Hex(low_byte));
    } //End Hex
    else if (k == 'f') //Print Float
    {
        rs_out('!');
    } //End Float
    else if (k == 'u') //Print Direct Character
    {
        //All ascii characters below 20 are special and screwy characters
        if(my_byte > 20) rs_out(my_byte);
    } //End Direct

    } //End Special Chars

    else
        rs_out(k);
    }
}

//Bit banging functions for direct connect from pic to serial port comm
//No MAX232 chip required - dirty, but it works
//=====
void rs_wait(void)
{

#ifdef Clock_4MHz
    int i;
    //for(i = 0 ; i < counter ; i++);
    for(i = 0 ; i < 6 ; i++);
#endif

#ifdef HS_Osc
    int i;
    //for(i = 0 ; i < counter ; i++);
    for(i = 0 ; i < 36 ; i++);
#endif

}

//Sends nate out at 9600 Baud
void rs_out_bb(uns8 nate)
{
#ifdef Clock_4MHz
#pragma rambank 1
    int l;

```

```

Serial_Out_BB = 1;
delay_us(9);
//rs_wait();
nop();
nop();
nop();

for(l = 0 ; l < 8 ; l++)
{
    Serial_Out_BB = !nate.0;
    nate = rr(nate);
    delay_us(9);
    //rs_wait();
}

Serial_Out_BB = 0;
//rs_wait();

//6-20-02 Needed a long wait time. Stuff was getting messed up
delay_us(10);
//delay_ms(1);

#endif

#ifdef HS_Osc
#pragma rambank 0
    int l;

    Serial_Out_BB = 1;
    rs_wait();

    for(l = 0 ; l < 8 ; l++)
    {
        Serial_Out_BB = !nate.0;
        nate = rr(nate);
        rs_wait();
    }

    Serial_Out_BB = 0;
    //rs_wait();

    //6-20-02 Needed a long wait time. Stuff was getting messed up
    delay_ms(1);
#endif
}

//Prints a string including variables
void printf_bb(const char *nate, uns8 my_byte)
{
    //Display byte

#pragma rambank 0
    uns8 i, k, m;
    uns8 high_byte = 0, low_byte = 0;

```

```

uns8 y, z;

for(i = 0 ; ; i++)
{
    k = nate[i];

    if (k == '\0')
        break;

    else if (k == '%') //Print var
    {
        i++;
        k = nate[i];

        if (k == '\0')
            break;
        else if (k == 'b') //Print Binary
        {
            for( m = 0 ; m < 8 ; m++ )
            {
                if (my_byte.7 == 1) rs_out_bb('1');
                if (my_byte.7 == 0) rs_out_bb('0');

                if (m == 3) rs_out_bb(' ');

                my_byte = my_byte << 1;
            }
        } //End Binary
        else if (k == 'd') //Print Decimal
        {
            /*if (my_byte.7 == 1)
            {
                rs_out_bb('-');
                my_byte.7 = 0;
            }*/

            z = my_byte;

            y = z / 100;
            z = z % 100;
            if (y != 0)
                rs_out_bb(bin2Hex(y)); //Print 100s

            y = z / 10;
            z = z % 10;
            if (y != 0)
                rs_out_bb(bin2Hex(y)); //Print 10s

            rs_out(bin2Hex(z)); //Print 1s
        } //End Decimal
        else if (k == 'h') //Print Hex
        {
            high_byte.3 = my_byte.7;
            high_byte.2 = my_byte.6;
            high_byte.1 = my_byte.5;

```

```

        high_byte.0 = my_byte.4;

        low_byte.3 = my_byte.3;
        low_byte.2 = my_byte.2;
        low_byte.1 = my_byte.1;
        low_byte.0 = my_byte.0;

        rs_out_bb('0');
        rs_out_bb('x');

        rs_out_bb(bin2Hex(high_byte));
        rs_out_bb(bin2Hex(low_byte));
    } //End Hex
    else if (k == 'f') //Print Float
    {
        rs_out_bb('!');
    } //End Float

    else if (k == 'u') //Print Direct Character
    {
        if(my_byte > 20) rs_out(my_byte);
    } //End Direct

    } //End Special Chars

    else
        rs_out_bb(k);
    }
}

//Returns nate at 9600 Baud
int rs_in_bb(void)
{
#ifdef Clock_4MHz
    uns8 j, nate, counter = 0;

    //while (Serial_In == 0);
    while (Serial_In == 0 && counter < 250)
    {
        counter++;
        if (counter > 245) return(0);
    }

    for(j = 0 ; j < 8 ; j++)
    {
        nate.7 = !Serial_In;
        nate = rr(nate);
        rs_wait();
    }

    return(nate);
#endif
#ifdef HS_Osc

```

```

    uns8 j, nate, counter = 0;

    //while (Serial_In == 0);
    while (Serial_In_BB == 0 && counter < 250)
    {
        counter++;
        if (counter > 245) return(0);
    }

    for(j = 0 ; j < 8 ; j++)
    {
        nate.7 = !Serial_In_BB;
        nate = rr(nate);
        rs_wait();
    }

    return(nate);
#endif
}

```

10. 232_comm.c

```

#define HS_Osc
#define Serial_Out_BB  RB2
#define Serial_In_BB  RB7
#define Baud_9600

#include "d:\Pics\c\16F628.h" //Compiler specific header file
#include "d:\Pics\c\int16CXX.H" //General Interrupts header file

#pragma origin 4 //Mandatory when interrupts are used

#define TRUE          1
#define FALSE         0

bit print_it;
bit start_record;

uns8 data_in;

uns8 memory_array[64] @ 0xC0;
uns8 mem_spot;

interrupt serverX(void)
{
    int_save_registers
    char sv_FSR = FSR; // save FSR if required

    if(RCIF) //UART Recieve Interrupt
    {

```

```

    data_in = RCREG;

    print_it = TRUE;

    //No clearing RCIF, must clear RCREG
    RCREG = 0;
}

FSR = sv_FSR;          // restore FSR if saved
int_restore_registers
}

#include "d:\Pics\code\Delay.c"    // Delays
#include "d:\Pics\code\stdio.c"    // Software and Hardware based RS232 Signals

//#pragma config |= 0x3F10 //Internal Oscillator, CProtect off, WDT off
#pragma config |= 0x3F02 //HS Oscillator, CProtect off, WDT off

#define DI1 RB3 //On-board LED

void main()
{

    PORTA = 0b.0000.0000;
    TRISA = 0b.0000.0000; //0 = Output, 1 = Input

    PORTB = 0b.0000.0000;
    TRISB = 0b.0000.0010; //0 = Output, 1 = Input - RB1 on the 16F628 is RX of the UART

    uns8 x;

    DI1 = 1; //Turn on the on-board LED
    mem_spot = 0;
    uns16 button_monitor = 0;

    enable_uart_TX(0); //Turn on Transmit UART with TX Interupts Disabled
    enable_uart_RX(1); //Turn on Receive UART with RX Interupts Enabled

    while(1)
    {

        if (print_it == TRUE) //We have something to record
        {
            GIE = 0;

            print_it = FALSE;
            button_monitor = 0; //Reset the LED blinker

            //Pass the pressed key back to hyperterminal
            printf("Ascii: %h\n\r", data_in);
            printf("Key: %u\n\r", data_in);

            //Store the incoming data to the memory array
            memory_array[mem_spot] = data_in;

```

```

mem_spot++;
if (mem_spot > 64) mem_spot = 0;

//Dump the memory array to the hyperterminal screen
if (data_in == 'Z')
{
    rs_out(12); //Clear the hyperterminal screen
    printf("Local Memory Dump:\n\r", 0);
    for(x = 0 ; x < 64 ; x++)
        printf("%u ", memory_array[x]);
}

//Finish up
GIE = 1;

CREN = 0;
CREN = 1;
}

//Blinks the LED when the counter called button_monitor rolls over
if(button_monitor == 0xFFFF)
    DI1 ^= 1;

button_monitor++;
}
}

```