M5

UART to Serial Terminal

Although the PIC32 is an elegant and powerful microcontroller, it doesn't stand so tall when compared against a PC in terms of raw computing power! Sometimes it makes sense to use the feasability of the microcontroller to work as a sensor or an embedded controller of some type but still be able to communicate with a more powerful computer. Sometimes you might just want a quick visual display or a *printf* terminal for debugging. Whatever the reason, this tutorial will provide a basic guide to PIC32 / PC communication.

UART

The module used for such communication is the PIC's *Universal Asynchronous Receiver/Transmitter (UART)* pronounced "you-art". For those familiar with computer music, the MIDI protocol uses a UART to do its serial communication. For those familiar with the Arduino, the TX/RX pins on an Arduino are the two main UART pins. In its most basic form the UART hardware consists of a transmit (TX) line and a receive (RX) line. The software configures how fast data is sent (the baud rate) and the specifics of the protocol.

Baud Rate

Since the UART module is *asynchronous*, there is no external clock line like the synchronous protocols SPI or I2C. Instead, both communicating devices must have their own clock sources configured to the same frequency. As long as these clocks match one another, data should be transferred smoothly. On the PIC32, the UxBRG special function register configures this rate called the baud rate. Note that the *x* in *UxBRG* is a placeholder for the reference number of the UART that is being configured. Common baud rates include 4800, 9600, 19200, 57600, and 115200 bits/s. Different baud rates can be chosen by the PIC32 but you must always make sure both devices are very stable at the selected rate. Sometimes it's best to select a lower baud rate to reduce the chance of errors. Since there is no shared clock, the hardware is simpler but there is an increased chance for the clocks to be out of sync causing incorrect data to be interpreted on the receiving end. For this reason, lower baud rates like 9600 are very popular.

The equation to configure the baud rate using the UxBRG register is below:

Equation 21-1: UART Baud Rate with BRGH = 0

$$Baud\ Rate = \frac{F_{PB}}{16 \cdot (UxBRG + 1)}$$

$$UxBRG = \frac{F_{PB}}{16 \cdot Baud\ Rate} - 1$$

Note: F_{PB} denotes the PBCLK frequency.

8N1

Like with most peripherals, the configuration options can be overwhelming! We'll be configuring our UART to use the standard *8N1* format. 8N1 meaning that each data trasfer consists of 8 bits of data, no parity bits, and 1 stop bit.

The parity bit provides an added layer of error detection. The parity bit is generated by the transmitting unit through a calculation done with the data being sent. If the receiving unit then calculates the same parity bit from the received data, it is assumed that everything is A-O-K. If a different parity bit is calculated, an error is triggered. For example, if odd parity checking is used, the transmitting unit would add up all "1" bits from the data being sent and set the parity bit to 1 if they are odd (like 01001100) or to 0 if even (like 01001101). The receiving unit would use the same calculation and compare the received parity bit to the calculated parity bit. Note that parity checking is done in hardware by the UART unit. However, once a parity error is flagged by the hardware, the programmer must deal with the error in software. Since parity checking will only take you so far (two corrupted bits is treated as no error!), you may desire something more powerful like a *Cyclic Redundancy Check (CRC)* which is outside the scope of this tutorial.

The stop bit signifies the end of a data transfer. There will almost always be either one or two stop bits. If the stop bits read by the receiving unit don't match those expected, the receiver will assume its clock drifted out of phase and a the software must deal with this error. It is also implied that a start bit is being sent as well with each transmission. The start bit allows the clock on the receiving end to do a *phase lock* with the data being received. Otherwise, the receiving end might end up sampling on the data transition edges instead of right in the middle of each bit. The data received would then look like utter nonsense!

So, in an 8N1 configuration, each byte of data sent is transmitted in the form of a 10-bit frame with the addition of one start bit, one stop bit, and no parity bits.

The example below configures the PIC32's UART1 module to an 8N1 format with a baud rate of 19200 bits/s and enables both transmit and receive functions (in *full duplex* mode).

```
int FPb = 40000000;       // peripheral bus clock frequency set at max SYSCLK
int desired_baud = 19200;
U1MODE = 0;               // disable autobaud, TX and RX enabled only, 8N1, idle=
U1STA = 0x1400;           // enable TX and RX
U1BRG = (FPb / (16*desired_baud)) - 1

U1MODESET = 0x8000;     // enable UART1
```

Note that in the example above, we are using the standard baud mode set when *UxMODE<3> = 0*. The UART clock is only able to take on specific frequencies and this bit helps determine how close the desired baud rate will be to the actual baud rate. When BRGH = 1 (i.e. UxMODE<3> = 1), high-speed mode is activated and the actual baud rate will probably be closer to the desired baud rate. However, each bit received on the UxRX pin will only be sampled once. When BRGH = 0 (i.e. UxMODE<3> = 0), multiple samples are taken on the UxRX pin to determine whether a HIGH or LOW is present. Even though the high-speed mode may have a more accurate baud rate clock, in standard-mode there is less chance to make receiving errors. See the UART reference manual section 21.6 for more information.

The following table provides a set of desired baud rates, the value you would set BRGH to for this baud rate and the error.

Table 21-2: UART Baud Rates (UxMODE.BRGH = '0')

Target Baud Rate	Peripheral Bus Clock: 40 MHz		
	Actual Baud Rate	% Error	BRG Value (decimal)
110	110.0	0.00	22726.0
300	300.0	0.00	8332.0
1200	1200.2	0.02	2082.0
2400	2399.2	-0.03	1041.0
9600	9615.4	0.16	259.0
19.2 K	19230.8	0.16	129.0
38.4 K	38461.5	0.16	64.0
56 K	55555.6	-0.79	44.0
115 K	113636.4	-1.19	21.0
250 K	250000.0	0.00	9.0
300 K			
500 K	500000.0	0.00	4.0
Min. Rate	38.1	0.0	65535
Max. Rate	2500000	0.0	0

Finally, the code below will configure the PIC32 for basic communication with a computer serial terminal. Using the configuration given and some of the functions, it can very easily be adapted for a variety of purposes. We decided it would be suitable just to mock you…

```
/* This program configures the PIC32 for communication with a computer at 57600 b
 * When the communication link is set up, the program will simply take input text
 * from the user and mock him/her. It demonstrates the basics of UART data Rx/Tx.
 */

// Include Header Files
#include <p32xxxx.h>                // include chip specific header file
#include <plib.h>                   // include peripheral library functions

// Configuration Bits
#pragma config FNOSC = FRCPLL       // Internal Fast RC oscillator (8 MHz) w/ PLL
#pragma config FPLLIDIV = DIV_2     // Divide FRC before PLL (now 4 MHz)
#pragma config FPLLMUL = MUL_20     // PLL Multiply (now 80 MHz)
#pragma config FPLLODIV = DIV_2     // Divide After PLL (now 40 MHz)
                                    // see figure 8.1 in datasheet for more info
#pragma config FWDTEN = OFF         // Watchdog Timer Disabled
#pragma config ICESEL = ICS_PGx1    // ICE/ICD Comm Channel Select
#pragma config JTAGEN = OFF         // Disable JTAG
#pragma config FSOSCEN = OFF        // Disable Secondary Oscillator
#pragma config FPBDIV = DIV_1       // PBCLK = SYCLK

// Defines
#define SYSCLK 40000000L

// Macros
```

```
// Equation to set baud rate from UART reference manual equation 21-1
#define Baud2BRG(desired_baud)      ( (SYSCLK / (16*desired_baud))-1)

// Function Prototypes
int SerialTransmit(const char *buffer);
unsigned int SerialReceive(char *buffer, unsigned int max_size);
int UART2Configure( int baud);

int main(void)
{
    char   buf[1024];       // declare receive buffer with max size 1024

    // Peripheral Pin Select
    U2RXRbits.U2RXR = 4;    //SET RX to RB8
    RPB9Rbits.RPB9R = 2;    //SET RB9 to TX

    UART2Configure(57600);  // Configure UART2 for a baud rate of 57600
    U2MODESET = 0x8000;     // enable UART2

    /* Add a small delay for the serial terminal
     *  Although the PIC sends out data fine, I've had some issues with serial te
     *  being garbled if receiving data too soon after bringing the DTR line low
     *  starting the PIC's data transmission. This has ony been with higher baud
    int t;
    for( t=0 ; t < 100000 ; t++);

    SerialTransmit("Hello Earthling!\r\n");
    SerialTransmit("Talk to me and hit 'enter'. Let the mocking begin!\r\n\r\n");

    unsigned int rx_size;

    while( 1){
        rx_size = SerialReceive(buf, 1024);     // wait here until data is receiv
        SerialTransmit(buf);                    // Send out data exactly as recei

        // if anything was entered by user, be obnoxious and add a '?'
        if( rx_size > 0){
            SerialTransmit("?\r\n");
        }
        SerialTransmit("\r\n");

    }//END while( 1)

    return 1;
} // END main()

/* UART2Configure() sets up the UART2 for the most standard and minimal operation
 *  Enable TX and RX lines, 8 data bits, no parity, 1 stop bit, idle when HIGH
 *
 * Input: Desired Baud Rate
 * Output: Actual Baud Rate from baud control register U2BRG after assignment*/
int UART2Configure( int desired_baud){

    U2MODE = 0;         // disable autobaud, TX and RX enabled only, 8N1, idle=HI
    U2STA = 0x1400;     // enable TX and RX
    U2BRG = Baud2BRG(desired_baud); // U2BRG = (FPb / (16*baud)) - 1

    // Calculate actual assigned baud rate
    int actual_baud = SYSCLK / (16 * (U2BRG+1));
```

```
    return actual_baud;
} // END UART2Configure()

/* SerialTransmit() transmits a string to the UART2 TX pin MSB first
 *
 * Inputs: *buffer = string to transmit */
int SerialTransmit(const char *buffer)
{
    unsigned int size = strlen(buffer);
    while( size)
    {
        while( U2STAbits.UTXBF);    // wait while TX buffer full
        U2TXREG = *buffer;          // send single character to transmit buffer

        buffer++;                   // transmit next character on following loop
        size--;                     // loop until all characters sent (when size
    }

    while( !U2STAbits.TRMT);        // wait for last transmission to finish

    return 0;
}

/* SerialReceive() is a blocking function that waits for data on
 *  the UART2 RX buffer and then stores all incoming data into *buffer
 *
 * Note that when a carriage return '\r' is received, a nul character
 *  is appended signifying the strings end
 *
 * Inputs:  *buffer = Character array/pointer to store received data into
 *          max_size = number of bytes allocated to this pointer
 * Outputs: Number of characters received */
unsigned int SerialReceive(char *buffer, unsigned int max_size)
{
    unsigned int num_char = 0;

    /* Wait for and store incoming data until either a carriage return is receive
     *   or the number of received characters (num_chars) exceeds max_size */
    while(num_char < max_size)
    {
        while( !U2STAbits.URXDA);   // wait until data available in RX buffer
        *buffer = U2RXREG;          // empty contents of RX buffer into *buffer p

        // insert nul character to indicate end of string
        if( *buffer == '\r'){
            *buffer = '\0';
            break;
        }

        buffer++;
        num_char++;
    }

    return num_char;
} // END SerialReceive()
```

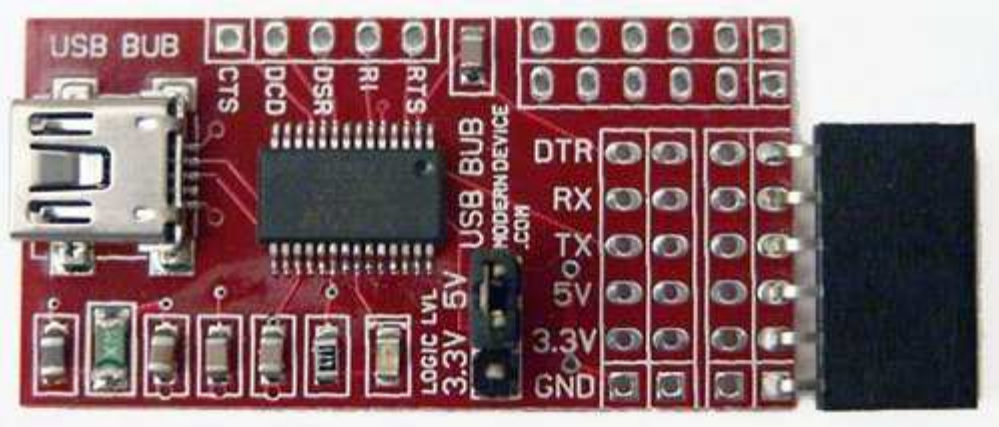
But how do we have late night conversations with this UART?

Although talking directly to your plant might help it grow, humans didn't evolve alongside microcontrollers and communication is a little bit trickier. There are plenty of ways to go from a computer to the asynchronous serial. Due to the near ubiquity of USB, our example will be USB based and require some conversion hardware.

Hardware

You will need to convert from USB with all of its complexity to a much simpler asynchronous serial signal. Two of the required data lines are obvious: the receive (Rx) and transmit (Tx) lines. We'll also want power in the form of a 3.3V supply somehow. Finally, we'll need a reset line so when the computer initiates a communication link, the PIC32 will be reset and no transmissions from the PIC will be lost.

I used a USB BUB from the folks down at Modern Device in Providence, RI for $14. Modern Device now sells a newer version aptly named the USB BUB II that works just as well.

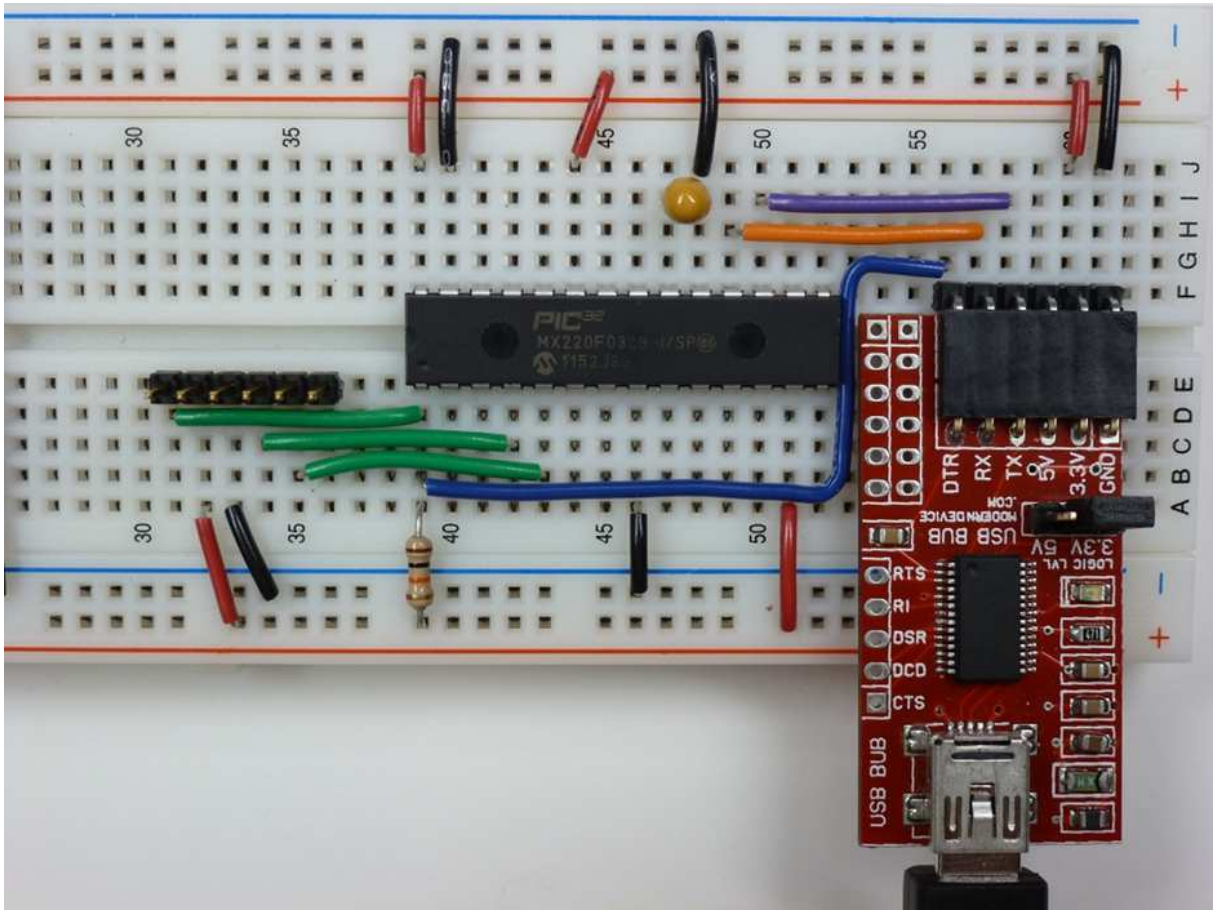


Another popular option would be to use a 3.3V TTL to UART cable made by FTDI like the TTL-232R-3V3 available for $20.

Whichever option you use, it will probably contain an FTDI chip of some sort. Also, make sure it supports 3.3V or you'll be logic level converting later.

Now connect the converter's Rx pin to the PIC's designated Tx pin (RB9 in the code) and the converter's Tx pin to the PIC's designated Rx pin (RB8 in code). Also, connect the converter's DTR or RTS line to the PIC's reset pin (MCLR). The DTR/RTS line will go LOW and then HIGH when the serial terminal initiates a communication link. This forces the PIC to reset and begin executing it's code from bootup.

Make sure you don't power the PIC with both the PICKit3 and your USB to UART converter! Differences VDD/GND levels will cause potentially large unwanted currents on the power rails. Our hardware setup with the BUB is shown below. Note how the BUB's jumpers are set to 3.3V operation instead of 5V.



Software

Although we no longer need to communicate via asynchronous serial data directly to the PIC, we now must send and receive USB packets to our go-between-device. Fortunately, this is pretty simple.

For **Windows** users, you can use PuTTY and select 'serial' as the type of communication. Then find the COM port number by going to *Device Manager -> Ports* after the hardware is connected. The baud rate we have chosen in the code example is 57600.

For **Mac** and **Linux** users, I've found GNU Screen to be an incredibly useful program. Not only can you use it as a serial terminal but it also allows you to multiplex between different terminal sessions in a single space! Goodbye screen clutter!

First, to find the device name we'll be communicating with, open a terminal and enter the command:

```
ls /dev | grep usb
```

For Linux users, the output will be something like:

```
ttyUSB0
```

where the number (0 above) will increase based on how many USB devices are connected.

For Mac users, the output will be something like:

```
cu.usbserial-A800cBEp
tty.usbserial-A800cBEp
```

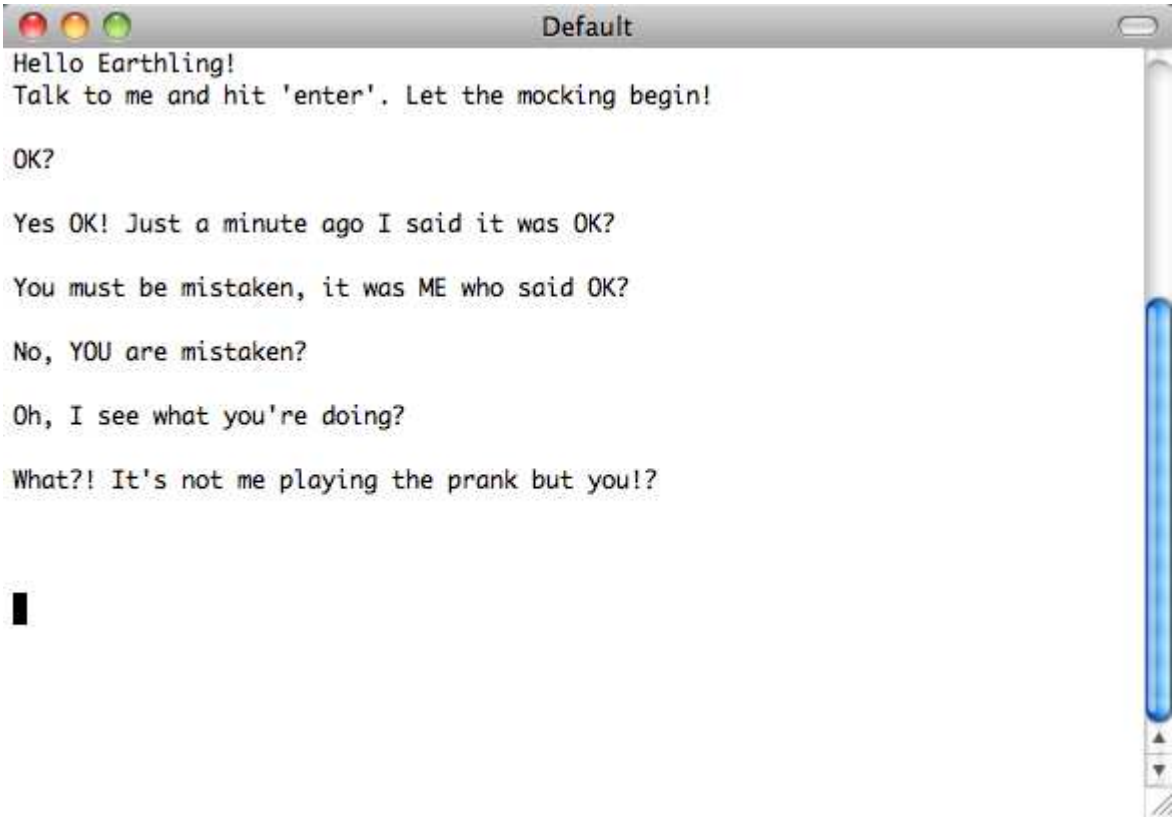
where both names reference the same USB device. Crudely, the 'tty' device reference expects a hardware assertion. Communication will be blocked until some reset line (DCD/DTR) has been asserted. The 'cu' device reference will be open for communication even without this initialization. Since we're using a DTR line through the USB BUB, the 'tty' name is used below although the 'cu' name should work fine.

If multiple devices are listed, disconnect/reconnect and run the list device command as needed.

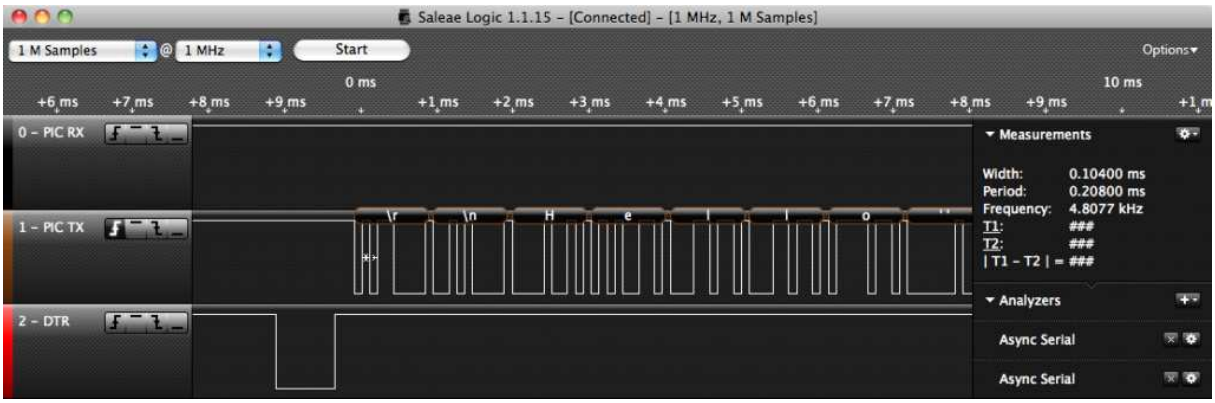
Once the PIC is programmed, the hardware all set, and the device name is known, use the following command to open a communication channel. Note that the program specifies a baud rate of 57600. Feel free to use the 'tab' command to finish any typing any commands that only have one useful outcome. That is, `tty.usb[tab]` is faster than typing `tty.usbserial-A800cBEp` if there's only one USB device connected.

```
screen /dev/tty.usbserial-A800cBEp 57600
```

To exit Screen, type `ctrl+a k y`. To see more Screen options, `ctrl+a ?`.



Below is a screenshot of the Saleae logic analyzer. Note that the DTR line is the reset signal sent from the computer to initiate communication with the PIC. The baud rate was set to 4800 for this example.



One thought on "UART to Serial Terminal"

12 November 2012 at 01:44

shalam

hi,
i am trying to sent my unsigned int RX=123456789 to uart.

```
unsigned int RX;
char str[16];

main(void)
{
mOSCSetPBDIV( OSC_PB_DIV_1 ); // Configure the PB bus to run at 1/1 the CPU
frequency
AD1PCFG = 0xFFFF; // Configure ALL pins as digital I/O

//SET UART PORTS DIRECTION
mPORTFSetPinsDigitalOut(BIT_5); // Make RF5 as output.

//UART2 CONFIG
OpenUART2(UART_EN | UART_NO_PAR_8BIT |
UART_1STOPBIT | UART_BRGH_SIXTEEN,
UART_TX_PIN_LOW | UART_TX_ENABLE, 86); //[80MHZ/(57600*16)]-1
U2MODESET = 0x8000; //ENABLE UART1
U2STASET=0x0400;

while(1)
{
RX=123456789;
utoa((int)RX, str, 10);
putsUART2("HELLO WORLD \r\n");

}
}
```

there's no error when i built it but nothing comes out to my hyperterminal when i run it.