

ECE 480 DESIGN TEAM 6

Interfacing a GPS Receiver with a Microprocessor via UART

Michael Carpenter

Dr. Shanker Balasubramaniam – Facilitator

Application Note

Friday, November 13, 2009

Executive Summary

For any sensing or data acquisition application, it is necessary to interface the processing module with the peripheral data acquisition device. Through UART (universal asynchronous receiver/transmitter) serial communication protocol, it is possible to effectively communicate between a microprocessor and a GPS module. This implementation facilitates efficient and reliable data transfer between the two devices.

Keywords: GPS, microprocessor, data transfer, UART, serial communication

Table of Contents

1. Introduction	3
2. Objective	3
3. Resources	3
4. Implementation and Result	4
4.1 Basic Circuit Connections	4
4.2 Programming.....	4
Configuring the UART.....	5
Monitoring the UART	6
Receiving Messages	7
Transmitting Messages	7
Example Program	8
5. Conclusion	10
6. References	10

1. Introduction

Interfacing peripheral data devices with a microprocessor is an important part of embedded product design. Microprocessors need to be able to acquire meaningful data from peripheral devices and also be able to send commands to those devices. An efficient and reliable means of transferring data between a microprocessor and a peripheral device can be achieved through serial UART communication. Using the Venus634LPx GPS receiver and the Microchip dsPIC30F3013 as an example, this application note will demonstrate an efficient and reliable communication method between a GPS receiver and a microprocessor.

2. Objective

The objective of this application note is to provide the reader with sufficient information on how to interface a GPS receiver with a microprocessor via UART. Specifically, this note will refer to interfacing a Venus634LPx GPS receiver with a Microchip dsPIC30F3013; however, the basic principles and procedures provided will allow for the interfacing of any modern GPS receiver to a feature sufficient microprocessor.

3. Resources

For the purposes of this application note, the reader should have the following resources available:

- Microchip MPLAB IDE
- Microchip ICD 2 or Microchip ICD 3 debugger
- Microchip dsPIC30F3013 Data Sheet
- Venus634FLPx Datasheet

It is assumed that the reader has a basic knowledge of simple circuit connections, MPLAB IDE, and microprocessor programming using a C compiler.

4. Implementation and Result

4.1 Basic Circuit Connections

In addition to the basic power connections needed for the GPS receiver, the microprocessor, and/or a crystal (please refer to the appropriate data sheets for this information), two connections need to be made for serial communication via UART between the receiver and the microprocessor. The TX pin on the receiver needs to connect to the U1RX pin on the microprocessor, and the RX pin on the receiver needs to connect to the U1TX pin on the microprocessor. This is shown in Figure 1 below. (Note: The U2TX and U2RX pins could be used as well; however, this application note will assume the U1TX and U1RX pins were chosen.)

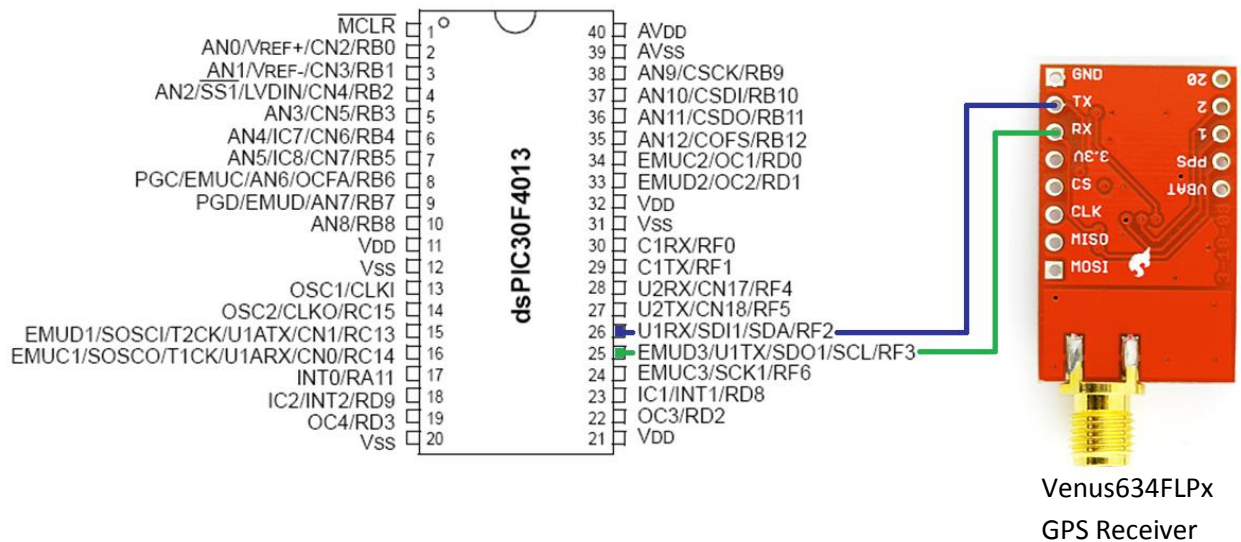


Figure 1

4.2 Programming

For serial UART communications between the microprocessor and the GPS receiver, the microprocessor needs to be configured for UART communication. The GPS receiver is already configured for UART communications and requires no external programming or set up to enable.

For ease of programming, it is important to include helpful header files that contain many basic functions that simplify the code. It may be necessary to download the appropriate files from manufacturers' websites. For this example, the following header files should be included from Microchip for the dsPIC30F4013:

- p30f4013.h
- uart.h

The dsPIC30F4013 has several control registers that configure its UART operation. These registers must be properly set for UART to function properly.

Configuring the UART

U1MODE: UART1 Mode Register

The U1MODE register contains several important bits that need to be properly set to enable the U1RX and U1TX lines to receive and transmit bits.

UARTEN: UART Enable bit

1 = UART is enabled.

0 = UART is disabled.

This bit needs to be set to 1 to enable UART communication in the program. This can be done with the following line of code:

```
U1MODEbits.UARTEN = 1;
```

PDSEL<1:0>: Parity and Data Selection Bits

11 = 9-bit data, no parity.

10 = 8-bit data, odd parity.

01 = 8-bit data, even parity.

00 = 8-bit data, no parity.

The configuration of these two bits depends on the GPS in use. The Venus634FLPx GPS receiver uses 8-bit communication with no parity, so a value of 00 should be set.

```
U1MODEbits.PDSEL = 0;
```

STSEL: Stop Selection bit

1 = 2 Stop bits.

0 = 1 Stop bit.

Similar to the PDSEL bits, this bit is dependent on the specific GPS chip. The Venus uses only one stop bit, so set this bit to 0.

```
U1MODEbits.STSEL = 0;
```

The remaining bits in this register can remain at their default values and do not need setting for this specific application.

U1BRG Register

The U1BRG register is the UART baud rate generator. This control register determines the baud rate of the communication. If not set to the proper value, the UART register will fail to receive and transmit messages. There is a basic equation to determine the value that needs to be set.

$$U1BRG = \frac{F_{CY}}{16 \cdot \text{Baud Rate}} - 1$$

In the equation above, F_{CY} denotes the instruction cycle clock frequency. This information can be found in the microprocessor's data sheet. For this example, we will have F_{CY} equal 10MHz. The desired baud rate is determined by the GPS receiver. The Venus receiver has a default baud rate of 9600.

$$U1BRG = \frac{10 \times 10^6}{16 \cdot 9600} - 1 = 64.1042 \approx 64$$

Using the above equation, for the desired baud rate of 9600 and an instruction cycle frequency of 10MHz, the U1BRG register needs to be set to 64. (Note: the register does not hold fractional values, so rounding is necessary. Small percent errors will not have an adverse effect.) The value of 64 in decimal equals 40 in hexadecimal, so set the U1BRG register with the following line of code:

```
U1BRG = 0x0040;
```

The UART should now be configured for use!

Monitoring the UART

While not always necessary, it can be very useful for both debugging and programming purposes if the status of the UART is closely monitored. The U1STA register contains bits regarding the current status and state of the UART port. Some of the more important bits regarding this application are explained below.

U1STA: UART1 Status and Control Register

UTXEN: Transmit Enable bit

1 = UART transmitter enabled.

0 = UART transmitter disabled.

This bit needs to be set high in order for the microprocessor to be able to transmit messages.

```
U1STAbits.UTXEN = 1;
```

URXISEL<1:0>: Receive Interrupt Mode Selection bits

11 = Interrupt flag bit is set when Receive Buffer is full (i.e., has 4 data characters)

10 = Interrupt flag bit is set when Receive Buffer is ¾ full (i.e., has 3 data characters)

0x = Interrupt flag bit is set when a character is received

These bits determine when the interrupt flag is set high for received messages. It is generally advisable to monitor this bit to ensure that no data being received will be lost.

Once the receive buffer is full, all incoming messages will be lost, and they will be unrecoverable. Reading characters off of the buffer will remove them from the buffer and make room for more incoming data. For the example program, an interrupt flag is set high whenever a character is received.

```
U1STAbits.URXISEL = 0;
```

PERR: Parity Error Status bit (Read Only)

1 = Parity error has been detected for current character.

0 = Parity error has not been detected.

For the GPS, which contains a check sum in its messages instead of a parity bit, it is unnecessary to monitor this register. However, in cases where a parity bit is used, it is a useful tool for detecting errors in received messages.

```
parityError = U1STAbits.PERR;
```

FERR: Framing Error Status bit (Read Only)

1 = Framing Error has been detected for the current character.

0 = Framing Error has not been detected.

This bit is useful for monitoring if errors have occurred when receiving bits. If a framing error occurs, the data may be corrupt.

```
frameError = U1STAbits.UFERR;
```

OERR: Receive Buffer Overrun Error Status bit (Read/Clear Only)

1 = Receive buffer has overflowed.

0 = Receive buffer has not overflowed.

This is an important register to monitor. If this bit is high, it means that all future incoming data is lost and unrecoverable. Once high, this bit needs to be cleared to allow messages to be received again. Clearing this bit will also clear the receive buffer, so if the data in the receive buffer is desired, it should be read and stored before clearing.

```
if(U1STAbits.OERR == 1) //if overrun error has occurred  
U1STAbits.OERR = 0; //clear OERR bit to allow for future messages
```

URXDA: Receive Buffer Data Available bit (Read Only)

1 = Receive buffer has data, at least one more character can be read

0 = Receive buffer is empty

This bit indicates if data is ready to be read. It can be used as a check before trying to read in a character.

```
readyToRead = U1STAbits.URXDA;
```

Receiving Messages

Once the UART port is configured, receiving data can be achieved by using a few simple commands. In this application, receiving data refers to the microprocessor reading in and storing NMEA messages created by the GPS receiver. Reading in characters from the receive buffer must occur one at a time. The command `getcUART1()` will retrieve a character from the receive buffer. A code snippet to read and store a single character is shown below.

```
char c; //declare a char variable c  
  
if(U1STAbits.URXDA) //check to see if data is ready to be read  
{  
    c = getcUART1(); //grab the next character off of the receive buffer  
}
```

Transmitting Messages

The microprocessor will need to be able to send commands to the GPS receiver to configure the receivers output, such as types of NMEA messages outputted and refresh rates. The `WriteUART1(0xXX)` command will transmit a character of data on the UART TX line to the GPS receiver. For transmitting successive characters of data, it is important to include a wait

statement between each write statement—this is because the write statement takes longer to complete than an instruction in the code, so calling write statements repeatedly in succession will result in failed transmissions. An example code snippet is shown below.

```
U1STAbits.UTXEN = 1; //make sure the TX line is enabled

WriteUART1(0xA0); //transmit the character 0xA0
while(BusyUART1()); //wait until the first transmission is complete
WriteUART1(0xA1); //transmit the character 0xA1
while(BusyUART1()); //wait until the previous transmission is complete
WriteUART1(0x00); //transmit the character 0x00
while(BusyUART1()); //wait until the previous transmission is complete
```

Example Program

A simple fully functional example program is shown below. This program configures the UART port for use, sends a command to the GPS receiver through the TX line to configure the receiver's output, and then reads in and stores the last received GPS message. The program takes advantage of the fact that every NMEA message starts with the '\$' character, characters read in before the first '\$' are ignored.

```
#include <p30f4013.h> //include the proper header files
#include <string.h>
#include <uart.h>

_FOSC(CSW_FSCM_OFF & EC); //configure the oscillator for an external clock
//                          //(crystal)

int main()
{
    //initialize the UART port
    U1MODEbits.UARTEN = 1; //enable UART1
    U1MODEbits.PDSEL = 0; //8 bits, no parity
    U1MODEbits.STSEL = 0; //1 stop bit used
    U1BRG = 0x0040; //set baud rate to 9600 (instruction rate is 10MHz because
    //of external clock)
    U1STAbits.URXISEL = 0; //interrupt bit is set when a character is received

    char c='0'; //declare a character
    char data[80]; //declare a character array to store an entire message
    int count = 80; //declare an integer to be used as an index variable

    //transmit the command for the GPS to only output GGA and VTG NMEA messages
    U1STAbits.UTXEN = 1; //enable the TX line
    while(BusyUART1());
    WriteUART1(0xA0); //send 0xA0 to the GPS
    while(BusyUART1()); //wait for completed transmission
    WriteUART1(0xA1); //send next character
    while(BusyUART1()); //wait again, etc.
    WriteUART1(0x00);
    while(BusyUART1());
    WriteUART1(0x09);
```

```

while(BusyUART1());
WriteUART1(0x08);
while(BusyUART1());
WriteUART1(0x01);
while(BusyUART1());
WriteUART1(0x00);
while(BusyUART1());
WriteUART1(0x00);
while(BusyUART1());
WriteUART1(0x00);
while(BusyUART1());
WriteUART1(0x00);
while(BusyUART1());
WriteUART1(0x01);
while(BusyUART1());
WriteUART1(0x00);
while(BusyUART1());
WriteUART1(0x00);
while(BusyUART1());
WriteUART1(0x08);
while(BusyUART1());
WriteUART1(0x0D);
while(BusyUART1());
WriteUART1(0x0A);
while(BusyUART1());

//transmission of command is finished, now read in characters

while(1)    //forever read in and store the last NMEA message
{
    if(U1STAbits.OERR) //check to see if there's an overrun error
        U1STAbits.OERR = 0; //clear to allow messages to be received

    if(U1STAbits.URXDA) //check if data is ready to be read
    {
        c = getcUART1(); //get a character from the receive buffer

        while(BusyUART1()); //wait until UART is getting the character
        if(c == '$') //c == $, hex=0x0024
        {
            count = 0; //for indexing the character array
        }

        if(count < 80)
        {
            data[count] = c; //store the character into the data array
            count++; //increment the index
        }
    }
}

return 1;
} //end program

```

5. Conclusion

By communicating with a GPS receiver through UART, a microcontroller can receive data from the GPS receiver and send commands to configure its output. ECE 480 Design Team 6 has implemented this set up and is using it to gather and store GPS position data for its speed and distance sensor for skiers and snowboarders. Design Team 6 has its microcontroller configure the GPS to limit the output to only contain messages with relevant data, and then it reads in that data every time it is sent.

6. References

[1] Microchip dsPIC30F3014/4013 Data Sheet

<http://ww1.microchip.com/downloads/en/DeviceDoc/70138F.pdf>

[2] Microchip dsPIC30F Family Reference Manual

http://www.microchip.com/stellent/groups/techpub_sg/documents/devicedoc/en026096.pdf

[3] Venus634FLPx Data Sheet

http://www.sparkfun.com/datasheets/GPS/Modules/Skytraq-Venus634FLPx_DS_v051.pdf

[4] Binary Messages of SkyTraq Venus 6 GPS Receiver

http://www.sparkfun.com/datasheets/GPS/Modules/AN0003_v1.4.8.pdf