GPS Reader

In this project a PIC 18F452 will be used to get one sentence from a GPS and make that sentence available to another PIC. This is done because the GPS puts out a bunch of sentences every second. These sentences are each approximately 80 bytes long. The GPS outputs them as serial data at 4800 bps, NEMA standard data. This is a problem because the data comes at very slow rate in is asynchronous. The PIC will read the data coming from the GPS and then internal move it to a data section available to another PIC via I²C.

I²C is a common 2 wire communication that is a synchronous serial transfer. One of the main advantages to using I²C is that it is based on an addressable bus. This means that several devices slave devices can be connected to the same bus and be controlled by a single master. One of the 2 wires in the interface is a clock line, which is driven by the master device. The other line is a bidirectional data line.

When starting I²C communication the first byte sent after the start condition is what will be called the control byte. It is important to note that some devices refer to this as the address, which is because it is the address of the device in relation to the bus. The last bit of the control byte tells the addressed device if it should send or receive data. In this case the GPS reader will be an I²C slave device. It will respond to the I²C through an interrupt. When the GPS reader gets a read request it transmits it last complete GPS sentence. The GPS reader only stores a sentence if it is the Recommended Minimum Specific GPS/TRANSIT Data Figure 1 ($GPRMC tagged sentence).

Figure 1: RMC NEMA Tag Definition

<table>
<thead>
<tr>
<th></th>
<th>RMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GPRMC,hhmmss.ss,A,III.II,a,yyyyy.yy,a,x,x,x,dmmmyy,x,x,a*hh</td>
<td></td>
</tr>
</tbody>
</table>

RMC = Recommended Minimum Specific GPS/TRANSIT Data

1 = UTC of position fix
2 = Data status (V=navigation receiver warning)
3 = Latitude of fix
4 = N or S
5 = Longitude of fix
6 = E or W
7 = Speed over ground in knots
8 = Track made good in degrees True
9 = UT date
10 = Magnetic variation degrees (Easterly var. subtracts from true course)
11 = E or W
12 = Checksum

In Listing 1 the code that reads the GPS sentences from the serial line, moves it to the outgoing string, and handles the I²C slave transmission. The string is so long that it is
likely if not definite that a serial byte or bytes will be will be in the incoming buffer while the \textsuperscript{I}2\textsuperscript{C} slave is transmitting the outgoing string. To compensate for this after each byte is loaded into \textsuperscript{I}2\textsuperscript{C} module for transmission the program checks to see if there is a byte in the serial buffer. If there is it moves it to a buffer that the serial data can be read from after the \textsuperscript{I}2\textsuperscript{C} transmission is done. The program also checks the incoming data for the correct tag before loading it in to the receive buffer. The end of the sentence is determined by a Line Feed command (byte value of 10). After a sentence is completely received it is copied to the outgoing string and the program begins looking for the next one.

Listing 1: GPSslave.c

```c
#include <p18f258.h>
#include <delays.h>
#include <usart.h>
#include <i2c.h>
#include <string.h>

char  usartBuff[80], i2cBuff[80] = "hellol world ", inbuff[20],
      bufflength = 0, lastchar, temp, lastr = 0, nextw = 0;

// the tag array defines the NMEA sentence to be captured
char  tag[8] = "$GPRMC," , status = 0;

// high interrupts
void int_handle (void);

// tells the program where to find the interrupt code
#pragma code HIGH_INTERRUPT_VECTOR = 0x0
void high_ISR (void)
{
    _asm
        goto int_handle
    _endasm
}
#pragma code

// this command gets a byte from the receive register and places it into
// incoming serial buffer.
void getbyte(void){
    inbuff[nextw] = ReadUSART();
    // nextw is the location for the next received serial byte
    nextw ++;
    // if the next write location is beyond the buffer space go back
    // to the beginning of the buffer
    if (nextw >= 20)
        nextw = 0;
    // clear the serial received flag
    PIR1bits.RCIF = 0;
}

#pragma interrupt int_handle
void int_handle (void)
{
    char length = 0;
    INTCON = 0;
    PIR1 = 0;// clears all interrupts

    // if a I2C read is sent the 80 byte transmit buffer is sent
    // note the transmit buffer is initialized to “hello world”
    if (SSPSTATbits.R_W == 1){
```
for(length = 0; length < 80; length++){  
    // clear I2C flag  
    PIR1bits.SSPIF = 0;  
    // move byte to I2C Transmission register  
    SSPBUF = i2cBuff[length];  
    // releases the I2C clock line  
    SSPCON1bits.CKP = 1;  
    // check if there is a byte in the serial receive register  
    if(PIR1bits.RCIF)  
        getbyte();  
    // wait for the I2C module finish transmitting the byte  
    while ( !PIR1bits.SSPIF );  
}

// ignores any other I2C traffic  
if (SSPSTATbits.D_A == 1){  
    temp = SSPBUF;  
}

// ignores any other I2C traffic  
if (SSPBUF != 0x0F){  
    temp = SSPBUF;  
}

// ignores any other I2C traffic  
if (SSPCON1bits.SSPOV == 1){  
    SSPCON1bits.SSPOV = 0;  
    SSPSTATbits.BF = 0;  
    temp = SSPBUF;  
}

// release I2C clock line  
SSPCON1bits.CKP = 1;  
// re-enable interrupt  
INTCON = 0b11000000;

// this command gets the last byte from the serial receive buffer.  
// it also will wait for a byte to become available if there is non in the buffer  
int getlast(void){  
    char current;  
    // sets the current read position from the next read position (lastr)  
    current = lastr;  
    // increase the next read position  
    lastr++;  
    // check to see if the buffer needs to wrap around  
    if (lastr >= 20)  
        lastr = 0;  
    // if there is no data in the buffer get one  
    if (current == nextw){  
        while(!DataRdyUSART());  
        getbyte();  
    }
    // return the value from the serial receive buffer  
    return inbuff[current];
}

void main (void) {

OpenUSART(USART_TX_INT_OFF & // TX interrupt off
               USART_RX_INT_OFF & // RX interrupt off
               USART_ASYNCH_MODE & // Asynchronous mode
               USART_EIGHT_BIT & // 8-Bit
               USART_CONT_RX & // Continuous receive
               USART_BRGH_LOW, // Low Baud rate formula
               129); // set approximately 4800 bps

OpenI2C(SLAVE_7, SLEW_ON); // I2C slave mode
SSPADD = 0x0F; // sets I2C address to 0x0F
TRISC = 0xFF; // sets PortC as inputs
SSPCON1bits.CKP = 1; // releases the I2C clock line

// clears the incoming buffer
memset(usartBuff, 0, 80);

// set up interrupts for the I2C module and enable them
INTCON = 0b11000000;
INTCON2 = 0;
INTCON3 = 0;
PIE1 = 0b00001000;
PIE2 = 0;

// continuously repeat
while(1){
    lastchar = 0;
    status = 0; // the status variable tracks the
    // NEMA tag matching status

    while(status != 7) {
        lastchar = getlast(); // gets the last char out
        // out of the incoming
        // buffer

        if(tag[status] == lastchar) // checks the char with
            // NEMA tag
            status++;
        else
            status = 0; // if it doesn’t match
            // start matching over
            // again it not the right
            // tag
    }

    bufflength = 0; // used to terminated
    // string

    while(lastchar != 10){ // while receiving the
        // sentence

        // get a character from the buffer and put it into
        // the incoming buffer
        lastchar = getlast();
        usartBuff[bufflength] = lastchar;
        // increase the buffer length
        bufflength++;
    }

    // NULL terminate the string
    usartBuff[bufflength] = 0;

    // copies complete sentence to the outgoing buffer
    strcpy(i2cBuff, usartBuff);

    // clears the incoming buffer
memset(usartBuff, 0, 80);

In this program a preprocessor directive #pragma was used. This directive tells the compiler specifically where in the processor's memory where to put code or data. This is important since this code used interrupts. The 18 series PIC has the ability to set two priority interrupts (high and low) but in this code that option was not enabled. In this code when an interrupt is generated the processor jumps to 0x8 in the program memory then calls the interrupt handling subroutine. The “#pragma interrupt int_handle” also tells the compiler that the following code is the interrupt handler. All interrupt based code was derived from Example 4 in the MPLAB C18 C Compiler Getting Started guide.