SCI Communications GPS

Goal:

- Receive serial from a PC
- Read GPS sensors.

SCI Communications

A very common form of communications between computers is called *Serial Communications Interface (SCI)* or *Asynchronous Communications*. This is how your computer communicates to your PIC board when you download code. It is the basis for CAN communications in cars. It is how many GPS sensors communicate, It is how serial ports on computes work.

The idea behind SCI communications is how to transmit data using only a single wire (plus a common ground). In order to do this, several problems need to be solved.

First, there is no clock signal like we had in SPI data protocol. Because of this, the transmitter and receiver have to know ahead of time how long each bit lasts.

Second, there is no Chip Select line like SPI protocol has. To detect the start of a message, an extra 'start bit' is required. This tells the receiver that a data package is coming.

Third, since data can come at any time, if you are receiving SCI data, you almost have to use interrupts.

This lecture looks at

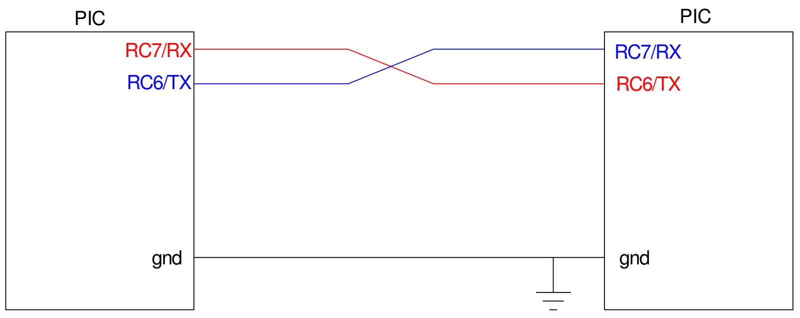
- How to receive SCI data using interrupts,
- How to receive numbers (such as 000 to 999) from another compuer or a keyboard, and
- How to read a GPS sensor.

Hardware Connection:

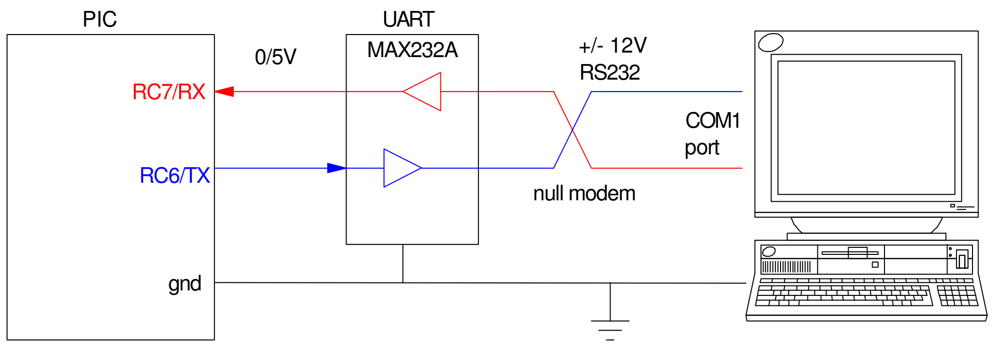
To communication with the serial port, you need three wires for 2-way communications (two wires for one-way communications):

- A common ground, and
- A transmit (TX) and Receive (RX) line.

Note that you need to swap the transmit and receive lines for data to get through.



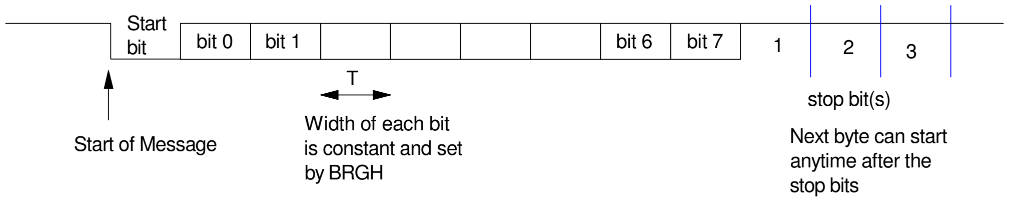
PIC to PIC communication via SCI communications



PIC to PC communication via SCI communications. Note that a UART is required to convert PIC voltage levels (0/5V) to RS232 voltage levels (±12V).

Also note that you need to swith the TX/RX lines between devices. This is what a null-modem does on a dB9 connector.

The data on an SCI bus looks like the following:


Start bit
bit 0
bit 1
bit 6
bit 7
1
2
3
T
Start of Message
Width of each bit is constant and set by BRGH
stop bit(s)
Next byte can start anytime after the stop bits

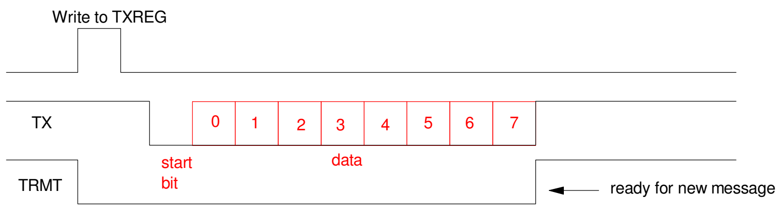
- When an SCI bus is not in use, it usually idles high (+5V). When a message is being sent, the line is pulled low signaling a byte follows.
- After the start bit, eight data bits are sent - least significant bit first.
- After the last bit is sent, the SCI bus returns to a high state, ready for the next data package.

With the PIC processor, you can both tranmit and receive SCI data.

SCI Transmission:

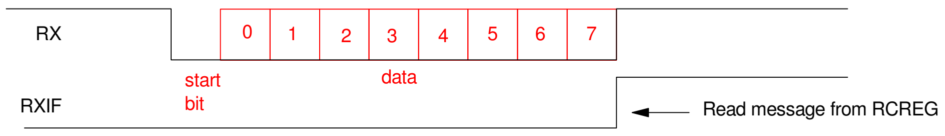
If you use the built-in SCI port, the timing for reading and writing the bits is automatic. For transmission:

- The hadware adds the start and stop bits
- The hardware sets up the timing for each bit.
- All you need to do in software is
  - Check that the SCI port is free (wait until TRMT=1)
  - Start the data transmission (write to TXREG)



SCI Reception: For reception

- The hardware detects the start bit automatically
- The hardware sets up the timing for reading in each bit
- The hardware actually reads each bit three times and does best-of-three voting to reduce errors
- The hardware then signals the software when 8-bits have beed read in by setting RCIF
- Once a bye has been recieved, the data can be read from RCREG.



How: Software:

In order to receive SCI data, the following steps need to be completed:

1. Set up PORTC as follows:

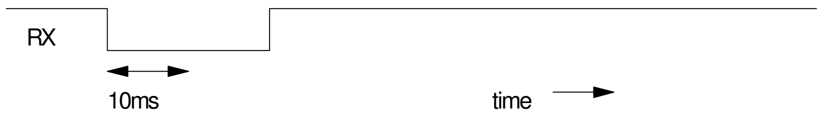
TRISC (address 0x__ - Bank __)

Bit	7	6	5	4	3	2	1	0
name	RX	TX	-	-	-	-	-	-
value	1	1	x	x	x	x	x	x

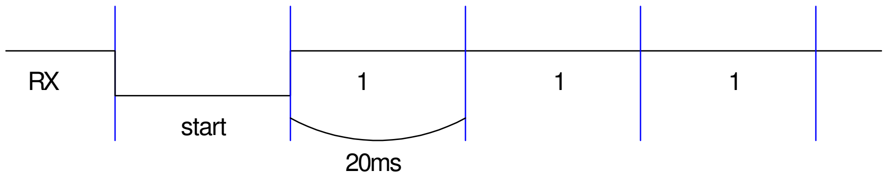
note: Both transmit and receive are set up as input.

- When TXEN=1 (transmit enable), you override TRISC and make RC6 an output.
- When TXEN=0 (transmit disabled), RC6 returns to high-impedance. This allows someone else to drive the data line.

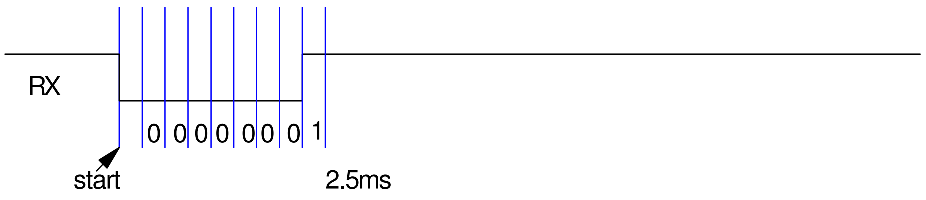
2. Set the baud rate. There is no clock, so the two devices **must** know how long each bit is. For example, suppose you recieve the following data:



If you think the data was transmitted at 50 baud (for one bit being 20ms), you'd read this as 0xFF



If you think the daa was transmitted at 4k baud (for one bit every 2.5ms), you'd read this as 0x80 (remember: LSB first)



Moral: The transmitter and reciever must be set to the same baud rate or else the data won't get through correctly.

The baud rate is set by bits SYNF, BRGH, and SPBRG

TABLE 10-1: BAUD RATE FORMULA (F_{OSC} = 10,000,00)		

SYNC	BRG16 = 1
	BRGH = 0
		BRG16 = 1
		BRGH = 1

0	Baud Rate = $F_{OSC}/(16(X+1))$	Baud Rate= $F_{OSC}/(4 (X+1))$

X = SPBRGH : SPBRG (0 .. 65,535)

Some common settings for a 20MHz crystal follow:

Baud Rate	SPBRG	BRGH	BRG16	SYNC	Error (%)
2,400	255	0	1	0	-1.70%
4,800	129	0	1	0	-0.16%
9,600	255	1	1	0	-1.70%
19,200	129	1	1	0	-0.16%
38,400	64	1	1	0	-0.16%
57,600	42	1	1	0	-0.95%
115,200	21	1	1	0	+1.44%

Transmit Status Register: TXSTA (address 0x98 - Bank 1)

Bit	7	6	5	4	3	2	1	0
Name	CSRC	TX9	TXEN	SYNC	-	BRGH	TRMT	TX9D
Write Vaue (for set-up)	1	0	1	0	-	1/0	-	-
Read Value	-	-	-	-	-	-	0/1	bit 9

bit 7: CSRC: Clock Source Select bit

- 1 = Clock generated by PIC
- 0 = Clock from external source

bit 6: TX9: 9-bit Transmit Enable bit

- 1 = Selects 9-bit transmission
- 0 = Selects 8-bit transmission

bit 5: TXEN: Transmit Enable bit

- 1 = Transmit enabled
- 0 = Transmit disabled

bit 4: SYNC: USART Mode Select bit

- 1 = Synchronous mode
- 0 = Asynchronous mode

bit 3: Unimplemented: Read as '0'

bit 2: BRGH: High Baud Rate Select bit

- 1 = High speed
- 0 = Low speed

bit 1: TRMT: Transmit Shift Register Status bit

- 1 = TSR empty
- 0 = TSR full

bit 0: TX9D: 9th bit of transmit data. Can be parity bit.

RCSTA: RECEIVE STATUS AND CONTROL REGISTER (ADDRESS 18h)

Bit	7	6	5	4	3	2	1	0
7ame	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
Write Vaue (for set-up)	1	0	1	1	0	-	-	-
Read Value (reception)	-	-	-	-	-	1/0	1/0	1/0

bit 7: SPEN: Serial Port Enable bit

- 1 = Serial port enabled (Configures RC7/RX/DT and RC6/TX/CK pins as serial port pins)
- 0 = Serial port disabled

bit 6: RX9: 9-bit Receive Enable bit

- 1 = Selects 9-bit reception
- 0 = Selects 8-bit reception

bit 5: SREN: Single Receive Enable bit (This bit is cleared after reception is complete.)

- 1 = Enables single receive
- 0 = Disables single receive

bit 4: CREN: Continuous Receive Enable bit

- 1 = Enables continuous receive
- 0 = Disables continuous receive

bit 3: ADDEN: Address Detect Enable bit

Asynchronous mode 9-bit (RX9 = 1)

- 1 = Enables address detection, enable interrupt and load of the receive burffer when RSR<8> is set
- 0 = Disables address detection, all bytes are received, and ninth bit can be used as parity bit

bit 2: FERR: Framing Error bit

- 1 = Framing error (Can be updated by reading RCREG register and receive next valid byte)
- 0 = No framing error

bit 1: OERR: Overrun Error bit

- 1 = Overrun error (Can be cleared by clearing bit CREN)
- 0 = No overrun error

bit 0: RX9D: 9th bit of received data (Can be parity bit)SSPEN = 1 to enable the SPI port.

Example 1:

Suppose you want to use your PIC to display the data on an SCI port. To do this,

1) Connect the serial line to RC7 (RX)

2) Set up the serial port to 9600 baud, with SCI interrupts turned on

```
// Turn on SCI interrupts at 9600 baud

   TRISC = TRISC | 0xC0;
   TXIE = 0;
   RCIE = 1;
   BRGH = 1;
   BRG16 = 1;
   SYNC = 0;
   SPBRG = 255;
   TXSTA = 0x22;
   RCSTA = 0x90;
```

3) Write an interrupt service routine which

- Reads in data from the PC
- Echos back each character as you type it in
- Saves the message in a buffer, and
- Looks for a carriage return <13> to terminate the message.

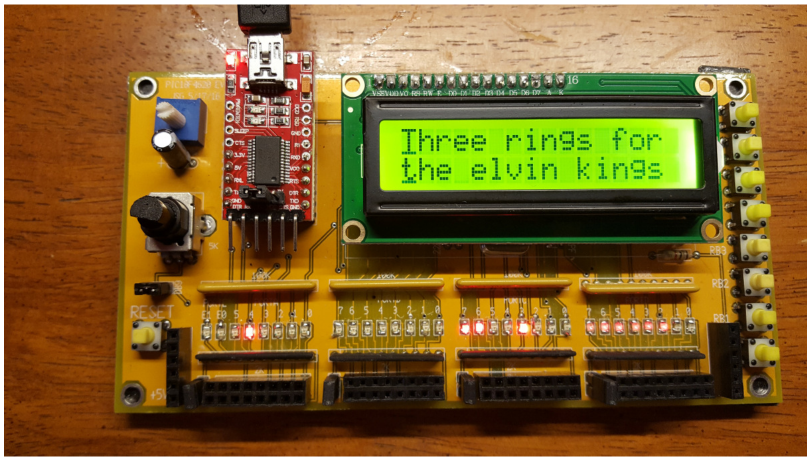
```
void interrupt IntServe(void)
{
   if (RCIF) {
      TEMP = RCREG & 0x7F;
      TXREG = TEMP;
      if (TEMP > 20) MSG0[M++] = TEMP;
      if (M > 21) M = 21;
      if (TEMP == 13){
         for (i=M+1; i<21; i++) MSG1[i] = ' ';
         for (i=0; i<20; i++) {
            MSG1[i] = MSG0[i];
            MSG0[i] = ' ';
            }
         M = 0;
         }
      RCIF = 0;
      }
   }
```

4) Now the main routine can display what's in the buffers MSG0 and MSG1. For example, if you type

Three rings for

the elvin kings

the following appears on the PIC board



Example 2: Type the numbers 000 to 999 followed by a carriage return. Have the PIC receive that number.

Solution: Inside the interrupt service routine, log the data coming in like before. When you see a carriage return (ascii 13). set a flag. This tells the main routine you just got some data.

```
void interrupt IntServe(void)
{
   if (RCIF) {
      TEMP = RCREG & 0x7F;
      TXREG = TEMP;
      if (TEMP > 20) MSG0[M++] = TEMP;
      if (M > 21) M = 21;
      if (TEMP == 13){
         FLAG = 1;
         for (i=M+1; i<21; i++) MSG1[i] = ' ';
         for (i=0; i<20; i++) {
            MSG1[i] = MSG0[i];
            MSG0[i] = ' ';
            }
         M = 0;
         }
      RCIF = 0;
      }
   }
```

Inside the main routine, watch for FLAG == 1. When that happens, parse the data into its 100's value, 10's value, and ones value:

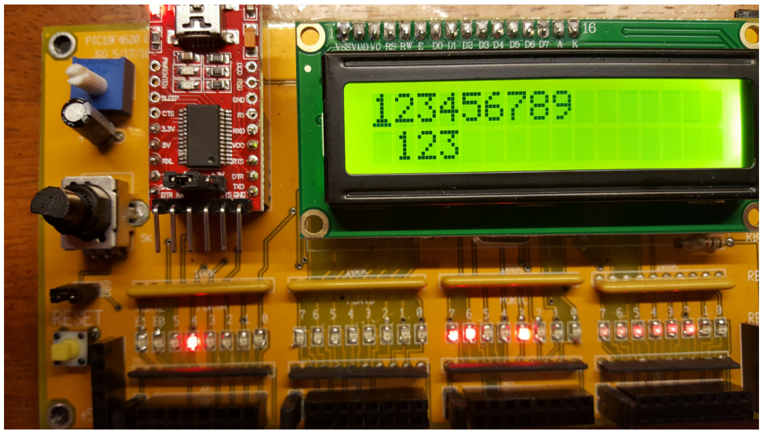
```
   while(1) {
      LCD_Move(0,0);
      for(i=0; i<16; i++) LCD_Write(MSG1[i]);

      if(FLAG) {
         DATA = (MSG1[0] - 48) * 100
              + (MSG1[1] - 48) * 10
              + (MSG1[2] - 48);
         LCD_Move(1,0);
         LCD_Out(DATA, 3, 0);
         FLAG = 0;
         }
```

```
            }
```

If you type in

```
     123456789 <cr>
```

you see the following:



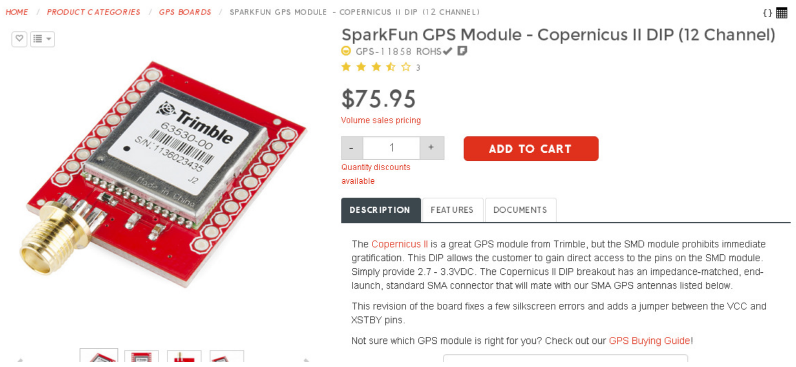
Here,

- The top row shows what stored in the buffer.
- The second row displays the number it converted this message into

Since only the first three digits are used, the number you get is 123. All of the other digits are ignored.

GPS Data

GPS communications also uses SCI protocol - typically 9600 baud. If you go to a site like SparkFun, you can buy GPS modules for less than $100


HOME / PRODUCT CATEGORIES / GPS BOARDS / SPARKFUN GPS MODULE - COPERNICUS II DIP (12 CHANNEL)
ADD TO CART
DESCRIPTION
FEATURES
DOCUMENTS

SparkFun GPS Module - Copernicus II DIP (12 Channel)

GPS-11858 ROHS✔

3

$75.95

Volume sales pricing

Quantity discounts available

The Copernicus II is a great GPS module from Trimble, but the SMD module prohibits immediate gratification. This DIP allows the customer to gain direct access to the pins on the SMD module. Simply provide 2.7 - 3.3VDC. The Copernicus II DIP breakout has an impedance-matched, end-launch, standard SMA connector that will mate with our SMA GPS antennas listed below.

This revision of the board fixes a few silkscreen errors and adds a jumper between the VCC and XSTBY pins.

Not sure which GPS module is right for you? Check out our GPS Buying Guide!

Connect it to an antenna, power, ground, and the serial line on your PIC board and the serial data will start flowing, looking something like this:

```
$GPGGA,152410.979,4731.42559,N,09233.10091,W,1,10,0.8,436.16,M,-30.59,M,
$GPGSA,A,3,15,05,08,29,27,18,21,26,06,22,,,1.4,0.8,1.1*30
$GPGSV,3,1,12,21,74,292,42,15,68,119,47,18,56,265,45,26,38,053,46*76
$GPGSV,3,2,12,48,25,230,23,29,25,190,39,06,24,310,39,27,18,120,42*7F
$GPGSV,3,3,12,03,15,319,,22,13,258,33,05,11,074,40,08,08,026,33*7D
$GPRMC,152410.979,A,4731.42559,N,09233.10091,W,0002.15,172.05,140312,,
$GPVTG,172.05,T,,M,0002.15,N,00003.98,K,A*08
$GPZDA,152410.979,14,03,2012,00,00*55
```

The challenge is then

- How to read in the GPS data into a buffer, and
- How to parse the data, and
- How to convert to meters.

Reading the GPS data into a buffer:

Already done . That was the Monitor program from before.

Parsing the data

The fields in the $GPRMC data mean the following:

```
$GPRMC,152410.979,A,4731.42559,N,09233.10091,W,0002.15,172.05,140312,,
```

Field	Data	Meaning
1	GPRMC	Recommended minimum GPS data
2	152410.979	Time: 15:23:10.979 UTC
3	A	A = OK, V = warning
4,5	4731.42559,N	Latitude: 47d 21.42559'
6,7	09233.10091,W	Longitude: 092d 33.10091'
8	0002.15	Speed (knots)
9	172.05	Direction of motion (degrees)
10	140312	Date: 14:03:12 March 14, 2012

This can then be converted to latitude and longitude:

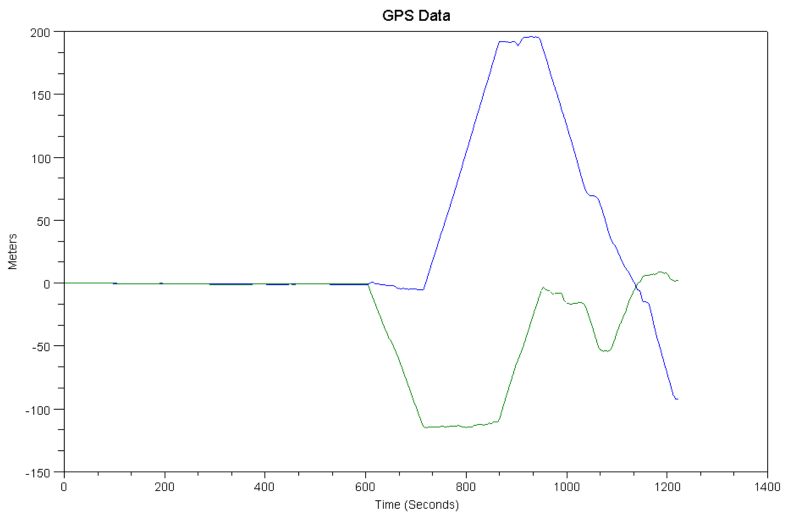
```
    // Latitude in minutes

          LATITUDE = (GPS[20] - 48)*600 +
                     (GPS[21] - 48)*60  +
                     (GPS[23] - 48)*10 +
                     (GPS[24] - 48)*1 +
                     (GPS[25] - 48)*0.1 +
                     (GPS[26] - 48)*0.01 +
                     (GPS[27] - 48)*0.001;

    // Longitude in minutes

          LONGITUDE = (GPS[33] - 48)*6000 +
                      (GPS[34] - 48)*600  +
                      (GPS[35] - 48)*60  +
                      (GPS[37] - 48)*10 +
                      (GPS[38] - 48)*1 +
                      (GPS[39] - 48)*0.1 +
                      (GPS[40] - 48)*0.01 +
                      (GPS[41] - 48)*0.001;
```

To convert to meters, assume the data was collected in Fargo, ND.

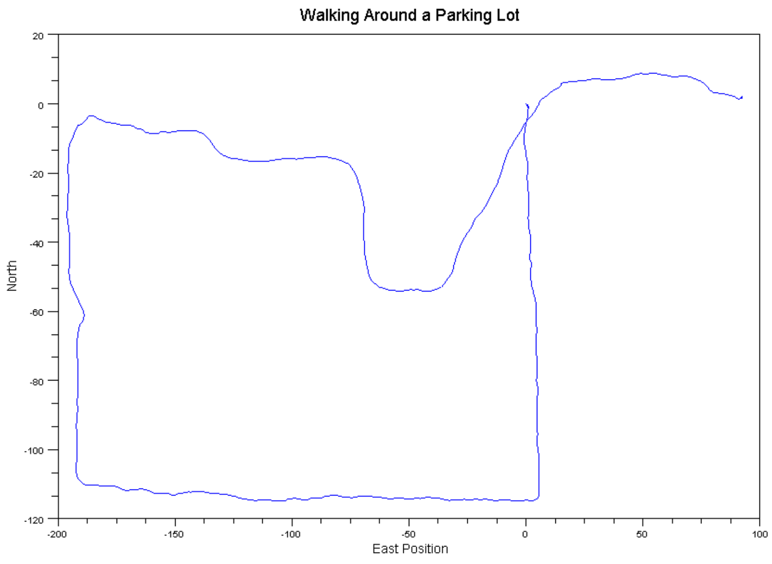
- Polar Radius = 6,356.8 km
  - 1 minute = 1849.12 meters
- Equatorial Radius = 6,378.1 km
  - 1 minute = 1,855.31 meters at the equator
  - 1 minute = 1,334.60 meters at 46 degrees north (Fargo)
- 1 Knot = 0.5144 meters/second

You can then log your position on Earth every second with an accuracy of a few inches.

As an example, GPS data was collected for 10 minutes as I walked around a parking lot. The position vs. time looks like the following:



If you plot X vs. Y, you can see your path over this 10 minute interval:



The challenge then is what to do with this ability. That's where engineering is a very creative field: we have a new toy that allows you to know your position on Earth within a few inches, up to 10 times a second. What do you do with it?