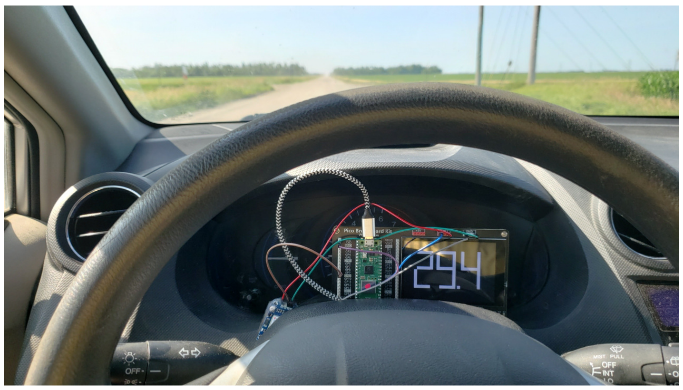
25. SCI Communications & GPS



Displaying a car's speed using a GPS sensor

Introduction:

Another sensor available for use with a Pi-Pico is a GPS sensor. These sensors use the global positioning sattelites to tell you your location on Earth incuding

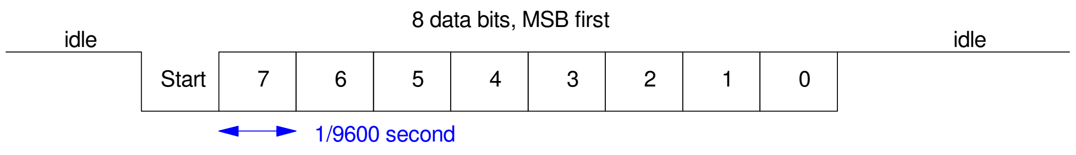
- Latitude
- Longitude
- Elevation
- Speed in knots, and
- Heading in degrees

With a GPS sensor, there's a wide variety of things you can do. This lecture looks at reading a GPS sensor and pulling out your position and speed. With that, we'll build adevice to

- Tell you where you parked your car, and
- Display your speed in mph on the graphics display.

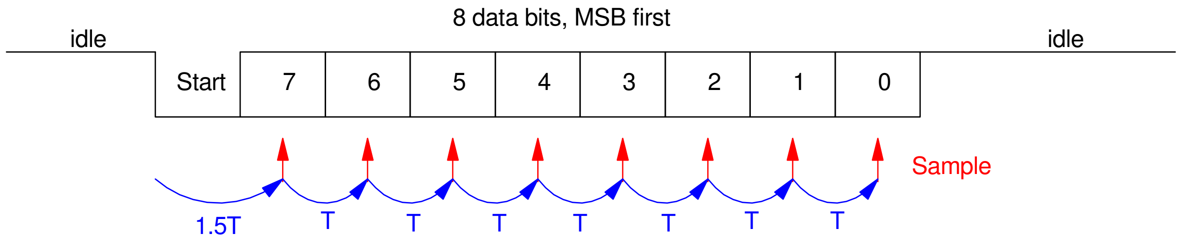
SCI Communication

GPS modules communicate using SCI communications. SCI (serial communications interface) is a type of asynchronous communications. With SCI communications, there is no clock and no chip select: only transmit and receive. Data is sent in 8-bit packets which are initiated with a start bit. Eight bits follow with a fixed bit length. For example, with 9600 baud communications, each bit would be 1/9600 second long.



SCI Write: Start bit comes first

When receiving serial data, the falling edge of the start bit indicates the start of a byte. The receiver then samples the middle of each bit to determine the message.



GPS Read: Sample each bit in the middle to determine its value.

With a Pi-Pico, you have two SCI ports available:

- SCI0:
  - TX=GP0, RX=GP1, or
  - TX=GP12, RX=GP13, or
  - TX=GP16, RX=GP17
- SCI1
  - TX=GP4, RX=GP5, or
  - TX=GP8, RX=GP9

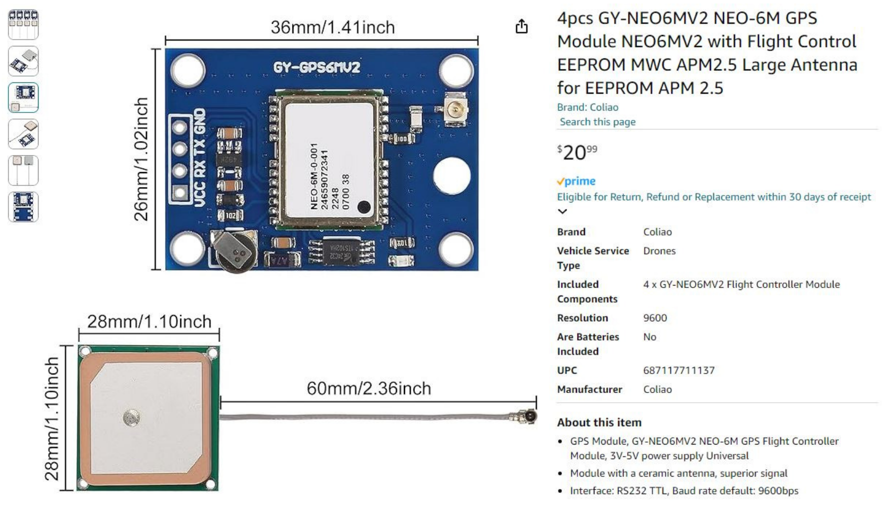
The procedure to initialize a SCI port is

```
from machine import UART

uart = UART(0, 9600)
uart.init(9600, bits=8, parity=None, stop=1, tx=0, rx=1)
```

Different ways to read and write to a UART are:

```
uart.read(5)            # read 5 characters into a buffer
uart.read()             # read all available characters
uart.readline()         # read a line (stop at carriage return)
uart.readinto(buf)      # read and store in a buffer
uart.write('Hello')     # write to the SCI port
```

GPS Modules & Messages



4pcs GY-NEO6MV2 NEO-6M GPS Module NEO6MV2 with Flight Control EEPROM MWC APM2.5 Large Antenna for EEPROM APM 2.5

Brand: Coliao

Search this page

$20.99

prime

Eligible for Return, Refund or Replacement within 30 days of receipt

Brand	Coliao
Vehicle Service Type	Drones
Included Components	4 x GY-NEO6MV2 Flight Controller Module
Resolution	9600
Are Batteries Included	No
UPC	687117711137
Manufacturer	Coliao

About this item

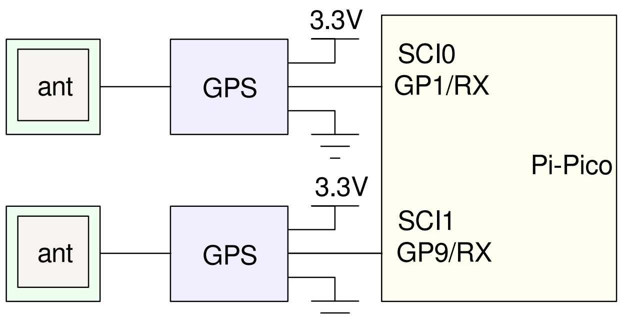
- GPS Module, GY-NEO6MV2 NEO-6M GPS Flight Controller Module, 3V-5V power supply Universal
- Module with a ceramic antenna, superior signal
- Interface: RS232 TTL, Baud rate default: 9600bps

GPS Sensor from Amazon

GPS sensors use satellites to tell you your location. They're actually really easy to use:

- Connect power and ground to the sensor
- Connect the serial out (TX) to the serial in (RX) on the Pi-Pico
- Receive serial data at 9600 baud (default).

Hardware: To connect a GPS receiver to your Pi-Pico, pick one of the two SCI modules. If you stick with the standard baud rate (9600 baud), only one wire is needed (plus power and ground of course).



Hardware for connecting a GPS module to a Pi-Pico (Only one GPS module is needed: your pick which SCI port is used)

Once the GPS module is connected to a Pi-Pico, the message can be read. For example, the following code reads a GPS receiver:

```
from machine import UART
from time import sleep

uart = UART(1, 9600)
uart.init(9600, bits=8, parity=None, stop=1, tx=8, rx=9)

while(1):
    x = uart.readline()
    sleep(0.2)
```

```
b'$GPGGA,205246.00,4649.55240,N,09652.11367,W,1,07,1.17,283.7,M,-27.5,M,,*69\r\n'
b'$GPGSA,A,3,23,18,10,27,15,32,24,,,,,,3.59,1.17,3.39*0C\r\n'
b'$GPGSV,2,1,08,08,19,311,09,10,52,288,24,15,28,055,21,18,47,147,25*78\r\n'
b'$GPGSV,2,2,08,23,77,015,19,24,39,100,21,27,32,277,1
b'$GPRMC,205247.00,A,4649.55258,N,09652.11395,W,0.306,,140724,,,A*62\r\n'
b'$GPVTG,,T,,M,0.306,N,0.567,K,A*22\r\n'
b'$GPGGA,205247.00,4649.55258,N,09652.11395,W,1,07,1.14,284.1,M,-27.5,M,,*6E\r\n'
b'$GPGSA,A,3,23,18,10,27,15,32,24,,,,,,2.49,1.14,2.22*04\r\n'
b'$GPGSV,2,1,08,08,19,311,08,10,52,288,25,15,28,055,22,18,47,147,26*78\r\n'
b'$GPGSV,2,2,08,23,77,015,19,24,39,100,21,27,32,277,1
b'$GPRMC,205248.00,A,4649.55297,N,09652.11403,W,0.312,,140724,,,A*63\r\n'
b'$GPVTG,,T,,M,0.312,N,0.578,K,A*29\r\n'
b'$GPGGA,205248.00,4649.55297,N,09652.11403,W,1,07,1.14,284.5,M,-27.5,M,,*6E\r\n'
b'$GPGSA,A,3,23,18,10,27,15,32,24,,,,,,2.49,1.14,2.22*04\r\n'
```

The different messages mean the following:

$GPGGA,205246.00,4649.55240,N,09652.11367,W,1,07,1.17,283.7,M,-27.5,M,,*69

- UTC time (hhmmss.sss). Data was recorded at 20:42:46.00 seconds GMT
- Latitude (ddmm.mmmm). 46 degrees, 49.55240 minutes north
- Longitude (ddmm.mmmm) 096 degrees 42.11367 minutes west
- Fix
  - 0: Fix not available or invalid
  - 1: GPS SPS mode, fixed valid
  - 2: Differential GPS, SPS mode, fix valid
- Satellites used (07, 1.17)
- Altitude (283.7 meters)

$GPGSA,A,3,23,18,10,27,15,32,24,,,,,,3.59,1.17,3.39*0C

- Satellites used in the position solution
- {3, 23, 18, 10, 27, 15, 32, 24}

$GPGSV,2,1,08,08,19,311,09,10,52,288,24,15,28,055,21,18,47,147,25*78

- The number of satellites in view

$GPRMC,205247.00,A,4649.55258,N,09652.11395,W,0.306,,140724,,,A*62

- Time, Data, Position, Course, and Speed
- A = valid data, V = invalid data
- Time (hhmmss.ss). Current time is 20:52:47.00

- Latitude (ddmm.mmmm). Location is 46 deg 49.55258 minutes north
- Longitude (ddmm.mmmm) Location is 096 deg 52.11395 minutes west
- Speed in knots: Speed is 0.306 knots
- Direction (in degrees)

Reading in a GPS Message

The Python command *uart.readline()* should in theory read each line of a GPS message, one by one. My experience is this command is inconsistent in capturing and separating each message. So, instead of using this line, custom subroutines were written using bottom up programming.

Level 1: GPS_Read_Line(chan):

The first step is to read each message into a string. The subroutine GPS_Read_Line

- Looks for a $ symbol to indicate the start of a message. If a second $ is read in, the previous message is discarded and a new message is begun
- It then reads in each byte from the serial port, byte by byte, into a string array
- The string array is terminated when a carriage return (ascii 13) is read.

This is placed inside a while() loop so that when called, the program will lock up in this subroutine until a GPS message is received.

```
def GPS_Read_Line(chan):
    flag = 0
    n = 0
    msg = ''
    while(flag == 0):
        if(chan == 0):
            x = uart0.read(1)
        else:
            x = uart1.read(1)
        if(x != None):
            x = ord(x)
            if(chr(x) == '$'):
                msg = ''
            if(x == 13):
                flag = 1
            else:
                msg = msg + chr(x)
    return(msg)
```

To test this subroutine, the results of a call to this subroutine are printed over and over:

```
while(1):
    msg = GPS_Read_Line(0)
    print(msg)
```

shell

```
$GPVTG,,T,,M,0.970,N,1.797,K,A*25
$GPGGA,173924.00,4649.55763,N,09652.11931,W,1,06,1.27,288.7,M,-27.5,M
,,*60
$GPGSA,A,3,29,18,15,13,20,23,,,,,,,2.52,1.27,2.18*0F
$GPGSV,4,1,16,01,21,283,17,05,55,055,07,07,02,027,,11,08,099,*7C
$GPGSV,4,2,16,13,42,105,19,15,45,156,12,16,11,324,,18,50,296,19*7F
$GPGSV,4,3,16,20,26,062,14,23,20,238,20,25,00,206,,26,16,289,*75
$GPGSV,4,4,16,29,64,190,19,30,03,056,,46,28,221,,48,30,216,*77
$GPGLL,4649.55763,N,09652.11931,W,173924.00,A,A*7F
$GPRMC,173925.00,A,4649.55729,N,09652.11947,W,1.143,,180724,,,A*67
$GPVTG,,T,,M,1.143,N,2.116,K,A*20
```

Level 2: String_to_Num()

Within each message are the numbers we want to pull out: latitude, longitude, and speed. These are in fixed fields within each line - assuming the line was read in correctly. For example assume a GPRMS message is read into a string, x:

```
x = '$GPRMC,173925.00,A,4649.55729,N,09652.11947,W,1.143,,180724,,,A*67'
```

The fields can be pulled out as:

$GPRMC,	173925.00	46	49.55729	096	52.11947	1.143
location	x[7:16]	x[19:21]	x[21:29]	x[32:35]	x[35:43]	x[46:51]
meaning	hhmmss GMT	latitude degrees	latitude minutes	longitude degrees	longitude minutes	speed knots

The python command *float()* should work - but if the string is read in wrong *float* crashes the program. To read in each string while avoiding a program crash, subroutine *String_to_Num()* was written.

For each element in string X:

- The value is checked if it is a number or a decimal
- If it's a number, the value being read is updated and stored in variable y
- If it's a decimal, a flag is set indicating that the following digits are decimal values
- If an invalid chacter is read, a global variable Error_Flag is set

```
def Str2Num(X):
    global Error_Flag
    n = len(X)
    y = 0
    flag = 0
    k = 0
    for i in range(0,n):
        z = X[i]
        if(z in {'0','1','2','3','4','5','6','7','8','9','0','.'}):
            if(z == '.'):
                flag = 1
            else:
                if(flag == 0):
                    y = 10*y + int(z)
                else:
                    k -= 1
                    y = y + int(z) * (10 ** k)
        else:
            Error_Flag = 1
    return(y)
```

To check this routine, several characters can be checked. For a valid number:

```
Error_Flag = 0
msg = '123.456'
print(Str2Num(msg), Error_Flag)
```

shell

```
123.456   0
```

For an invalid number:

```
Error_Flag = 0
msg = '1G3.456'
print(Str2Num(msg), Error_Flag)
```

shell

```
13.456   1
```

Level 3: GPS_Read(chan)

Once each line has been read and the fields can be parsed, the next level up pulls out each field within the GPS message. This routine looks for either a $GPRMC or a $GPGGA message. Once found, the latitude and longitude is read in. A flag is used to keep the subroutine looping until a valid message is received.

In the following code, the latitude and longitude are returned in units of degrees.

notes: You lose a little resolution by dividing the minutes by 60. In subsequent programs, only the minutes are used (the degrees are ignored). I'm not moving enough for the degrees to matter, so in my case, that doesn't hurt anything.

```
def GPS_Read(chan):
    flag = 0

    while(flag == 0):
        x = GPS_Read_Line(chan)
        if(len(x) > 52):
            if(x[3] == 'R'): # $GPRMC
                flag = 1
                time = Str2Num(x[7:16])
                LatD = Str2Num(x[19:21])
                LatM = Str2Num(x[21:29])
                LonD = Str2Num(x[32:35])
                LonM = Str2Num(x[35:43])
                speed = Str2Num(x[46:51])

    return([time, LatD + LatM/60, LonD + LonM/60, speed])
```

To test this routine, the field are pulled out of the GPRMC messages and displayed

```
while(1):
    [t, x, y, v] = GPS_Read(0)
    msg0 = str('{:9,0f}'.format(t) + ' ')
    msg1 = str('{:11.7f}'.format(x) + ' ')
    msg2 = str('{:11.7f}'.format(y) + ' ')
    msg3 = str('{:9.4f}'.format(v) + ' ')
    print(msg0 + msg1 + msg2 + msg3)
```

shell

```
183300  46.8258247  96.8686447    0.1620
183301  46.8258247  96.8686447    0.1150
183302  46.8258286  96.8686447    0.0290
183303  46.8258286  96.8686447    0.0490
```

Note:

- Column #1 (time) is incrementing by one. Each GPRMC message is being read (they come in once per second)
- Column #2 (latitude) is reading in as 46.8258247 degrees north (Fargo)
- Column #3 (longitude) is reading as 96.8686447 degrees west (Fargo)
- Column #4 (speed in knots) is reading in as zero-ish. The sensor isn't moving during this time.

At this point, everything looks good.

Where's My Car?

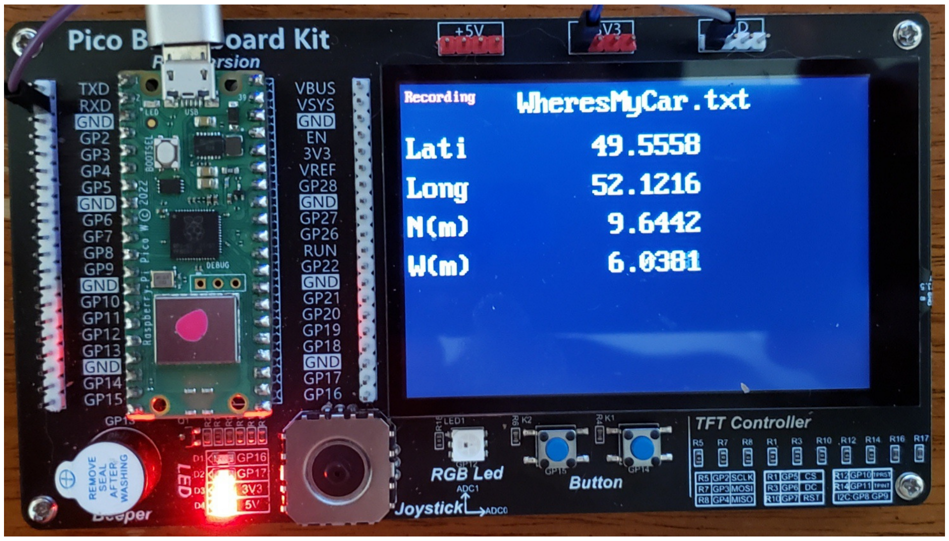
Level 4: Main Routine

- *this is a fairly long routine. Please see lecture #25 on Bison Academy for the listing*

Once it's confirmed that the GPS sensor is being read correctly, the main routine can be written. The following code

- Reads in a GPS sensor on UART0
- Rather than displaying the latitude and longitude, these are displayed in maters relative to a reference position

- Button GP15: When you press button GP15, the current position is recorded as the reference position. For example, when you park your car, press GP15. As you move about, your distance from your car is then displayed on the GPS.
- Button GP14: When you press button GP14, the data recording is toggled. When recording is turned on
  - a single beep is played, indicating that recording has started
  - a Recording message is displayed in the upper right corner of the screen
  - A data file is opened for appending, and
  - Data is recorded to that file
  - (the file name is defined at the start of the program)
- When recording is turned of
  - Two beeps are played indicating that recording to a file has been paused
  - The Recording message is cleared, and
  - The file is closed



TFT Display: Your latitude in minutes, longitude in minutes, and where you are relative to home position in meters GP15 resets the home position. GP14 toggles data recording.

The data file contains four columns of number

- Latitude in minutes
- Longitude in minutes
- Distance north of your home position in meters
- Distance west of your home position in meters

```
 Latutude    Longitude    North   West
 (minutes)   (minutes)     (m)     (m)
 49.5489616  52.1171951  0.6142  0.4879
 49.5489616  52.1171951  0.6142  0.4879
 49.5490990  52.1174164  0.8684  0.7680
 49.5489388  52.1175575  0.5719  0.9468
```

Text file for Wheres_My_Car.py. GPS position and distance to home position in meters

The conversion from latitude and longitude to meters is as follows:

The Earth's equatorial circumference is 40,075km (space.com). Each degree of longitude at the equator corresponds to a distance of 111.317km

$$1^0 = \frac{40{,}075km}{360} = 111.319km$$

Each minute corresponds to a distance of 1855.285m

$$1' = \frac{40{,}075km}{60 \cdot 360} = 1,855.324m$$

Scale by your latitude (assume 46.8258 degrees north)

$$1' = \left(\frac{40{,}075km}{60 \cdot 360}\right) \cdot \cos(46.8258^0) = 1,269.448m$$

So, in Fargo, one minute of longitude corresponds to 1269.448 meters east/west.

The Earth's polar circumference is 40,008km (space.com). Each degree of latitude corresponds to

$$1^0 = \frac{40{,}008km}{360} = 111.133km$$

Each minute of latitude corresponds to 1852.222 meters

$$1' = \frac{40{,}008km}{60 \cdot 360} = 1852.222m$$

With these conversions, if you know how far you are from home in terms of minutes of an arc, you can compute your distance in meters.

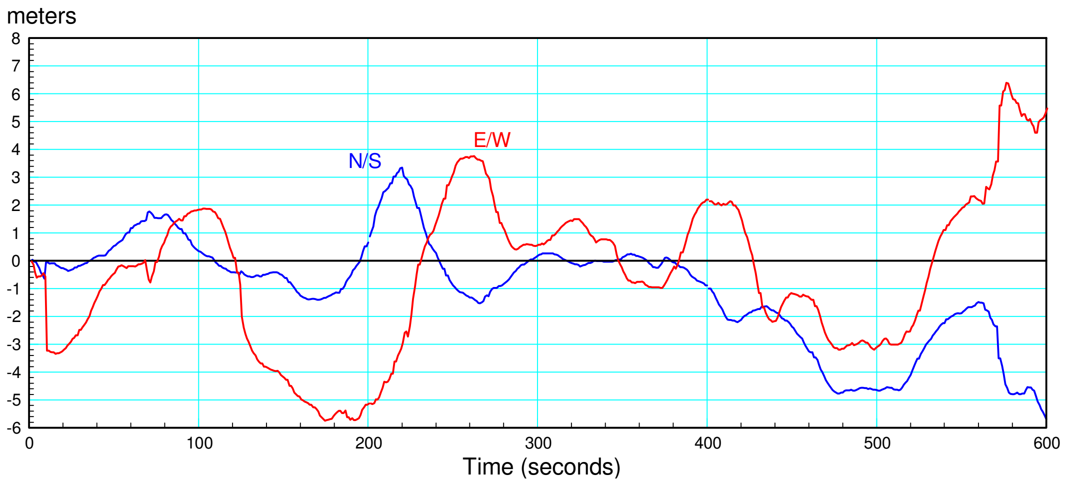
Noise on GPS Signals

Ideally, your GPS readings should be your position on the Earth. In practice, your GPS position will drift even when stationary. Disturbances in atmospheric conditions, reflections, etc. can cause your recorded GPS position to drift even when stationary.

To illustrate this, the previous program was used to measure the drift of a stationary GPS sensor:

- The GPS sensor was turned on for ten minutes
- The Home button was pressed (GP15) to record the current position, then
- The position of the GPS sensor over a span of 10 minutes was recorded (600 data points).

The resulting position reading is as follows:



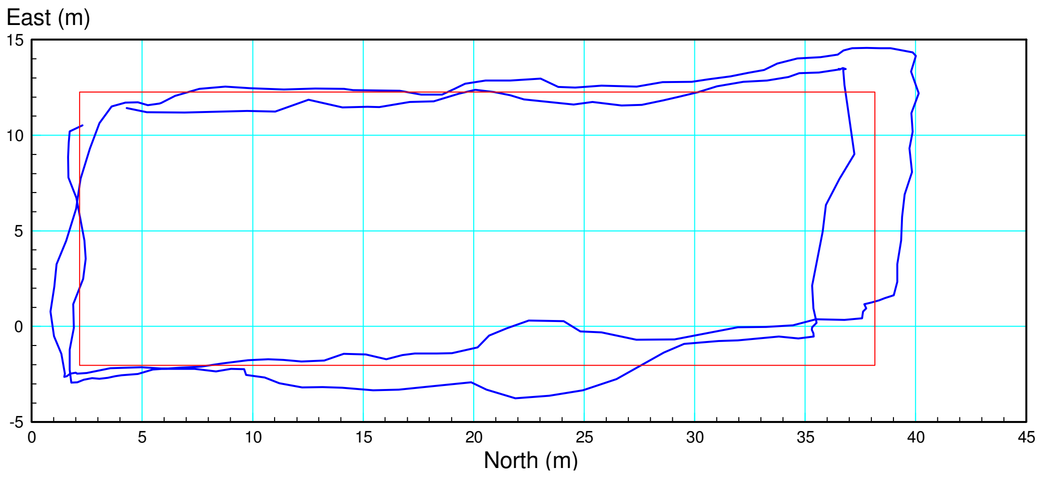
Drift in GPS signal for a stationary target

Even though the GPS sensor is stationary, the GPS reading is drifting. This allows you to estimate the certainty of the above GPS position: the 90% confidence interval is right around +/- five meters

	mean	st dev	1.66 * st dev
N/S	-1.025 m	1.897 m	3.15 m
E/W	-0.436 m	2.751 m	4.57 m

Based upon readings for a stationary GPS sensor, position is known within about 5 meters (3.15m and 4.57m)

This also shows up when the sensor is moving. The following plot shows the recorded GPS position when walking around a rectangle (shown in red).



Recorded GPS position when walking around a rectangle (blue) and actual position (red)

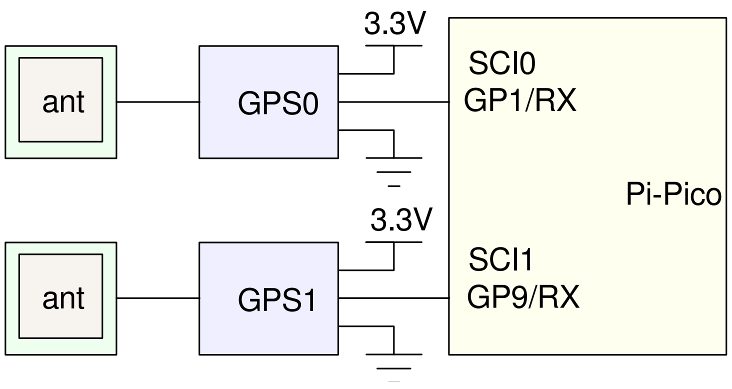
Differential GPS

One way to reduce the noise on a GPS sensor is to use differential GPS. The idea here is to have two GPS receivers:

- One is stationary with a known position
- The other GPS sensor is free to move about.

By taking the difference in the GPS readings, noise in the GPS signal should be greatly reduced. This assumes of course that the noise is highly correlated (i.e. both sensors see the same atmospheric disturbances, etc.)

In Python, this can be done by attaching two GPS receivers to your Pi-Pico. Presumably if one GPS is to move around, it would be connected through a wireless link. For now, both are connected directly.



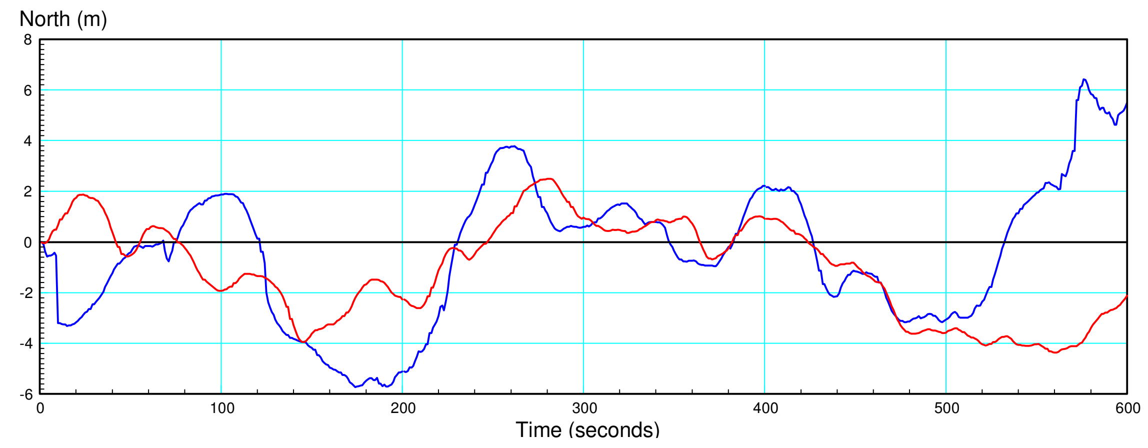
For differential GPS, two GPS receivers are used

In terms of software, both GPS receivers are read at each time point. Treating GPS0 as the reference signal, the location of GPS1 can be found by taking the difference in the two readings: (note: Error_Flag is set if there is an error in either GPS readings. The while-loop keeps going until both sensors give a valid position.)

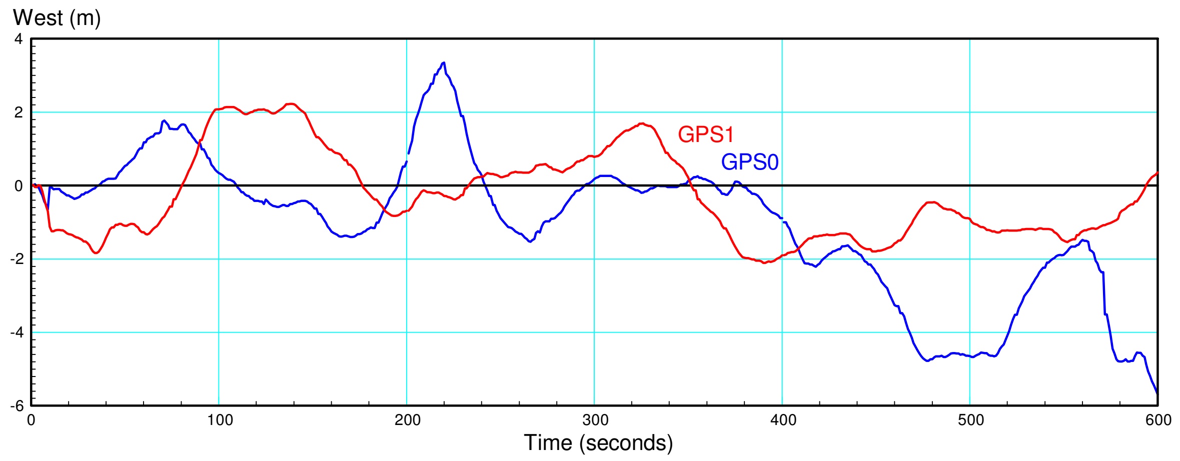
```
while(1):
    Error_Flag = 1
    while(Error_Flag == 1):
        Error_Flag = 0
        [t0, x0, y0, v0] = GPS_Read(0)
        [t1, x1, y1, v1] = GPS_Read(1)
    x = x1 - x0
    y = y1 - y0
```

Differential GPS code: The output is the position of GPS1 relative to GPS0

In theory, this should result in a large drop in the standard deviation of the GPS position (x,y). To check if this works, GPS readings for both channels were recorded for ten minutes with both modules stationary, 20cm apart.



Reported North/South location of two stationary GPS sensors



Reported East/West location of two stationary GPS sensors

Ideally, taking the difference in the reported GPS positions should give a better reading due to the noise canceling. This should show up in a reduction of the standard deviation. Actually, the standard deviation gets worse:

	std(GPS0)	std(GPS1)	std(GPS0-GPS1)
N/S	2.7507 m	1.9012 m	2.9768 m
E/W	1.8974 m	1.1917 m	1.9349 m

Standard deviation (spread) of GPS signals when stationary

The assumption with differential GPS is that the noise is highly correlated between the moving sensor (GPS1) and the stationary reference sensor (GPS0). The correlation coefficients between the two GPS signals can be found as (ECE 341 lecture #19):

$$cov(x, y) = E(xy) - E(x)E(y)$$

$$\rho_{x,y} = \frac{cov(x,y)}{\sigma_x \sigma_y}$$

In Matlab:

```
>> num = mean(x0 .* x1) - mean(x0)*mean(x1);
>> den = std(x0) * std(x1);
>> rhox = num / den

rhox =    0.2214

>> num = mean(y0 .* y1) - mean(y0)*mean(y1);
>> den = std(y0) * std(y1);
>> rhoy = num / den

rhoy =    0.2817
```

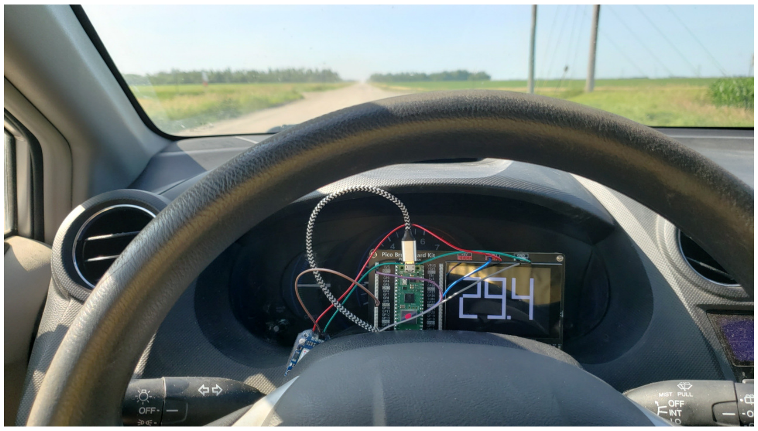
Translation: The two GPS sensor readings are *slightly* correlated (22% and 28%). This isn't enough to improve the GPS readings however:

- If the correlation was zero, the variances would add when you subtract the signals. This is property of normal distributions.
- If the correlation was one, the variances would subtract when you subtract the signals. A correlation of one means the signals are the same.

A correlation of 0.22 or 0.28 is somewhere in-between - but closer to a correlation of zero. Taking the difference actually made things worse.

Differential GPS does exist and it does work. Apparently, you need a better GPS sensor and/or antenna to make differential GPS work, however.

GPS Speedometer

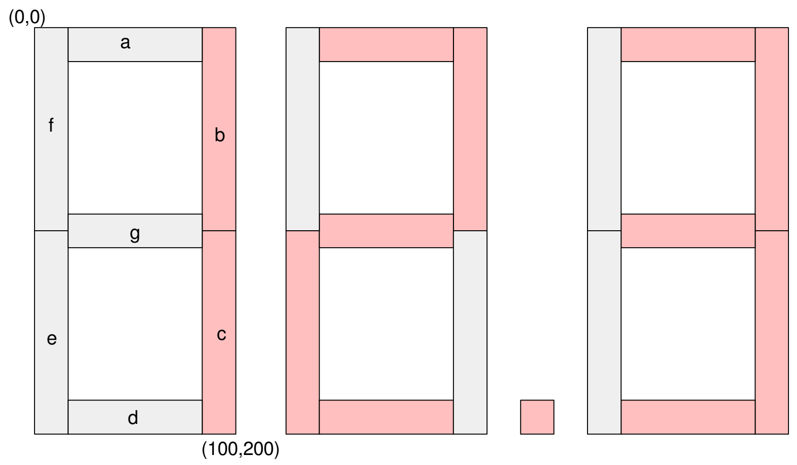


GPS Speedometer: A vehicle's speed is displayed using a GPS module

Finally, let's use the GPS sensor to measure your speed in mph. The $GPRMC returns your speed in knots. To convert to mph, multiply by 1.15077

$$mph = knots \cdot 1.15077$$

The fonts available in LCD.py are too small to see when driving, so let's also create a 100 x 200 font for numbers. This simply uses rectangles to mimic a 7-segment display



To make the display easier to read, 100 x 200 pixel numbers are used

The subroutine *Big_Display(N, x, y)* displays a single digit (0..9) using seven rectangles with the upper left corner being (x,y). It then draws a solid box for rectangles {a, b, c, d, e, f, g} either in white or black. This routine isn't fancy but works. With a little coding, it could probably be shortened considerable.

```
def Big_Display(N, x, y):
    T = 15
    c1 = LCD.RGB(250,250,250)
    c0 = 0
    if(N == 0):
        LCD.Solid_Box(x+T,y,x+100-T,y+T,c1)
        LCD.Solid_Box(x+100-T,y,x+100,y+100,c1)
        LCD.Solid_Box(x+100-T,y+100,x+100,y+200,c1)
        LCD.Solid_Box(x+T,y+200-T,x+100-T,y+200,c1)
        LCD.Solid_Box(x,y+100,x+T,y+200,c1)
        LCD.Solid_Box(x,y,x+T,y+100,c1)
        LCD.Solid_Box(x+T,y+100-T,x+100-T,y+100,0)
    if(N == 1):
        :
```

Your speed is displayed with the routine *Display(Speed).* This routine

- Pulls out the digits one by one (10s, 1's, 0.1s) then displays each digit on the TFT.

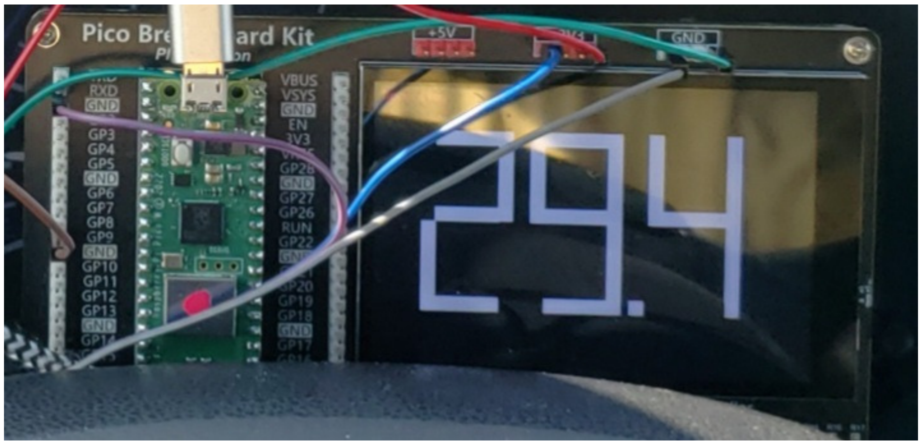
```
def Display(Speed):
    X = int(Speed*10)
    A0 = X % 10
    X = X // 10
    A1 = X % 10
    X = X // 10
    A2 = X % 10
    Big_Display(A2, 50, 50)
    Big_Display(A1, 170, 50)
    Big_Display(A0, 300, 50)
    LCD.Solid_Box(280,235,295,250,LCD.RGB(250,250,250))
```

The main loop then

- Reads the GPS sensor, picking out your speed in knots
- Checks button GP14 for turning recording on and off, and
- Displays your GPS speed on the LCD with a 100x200 font

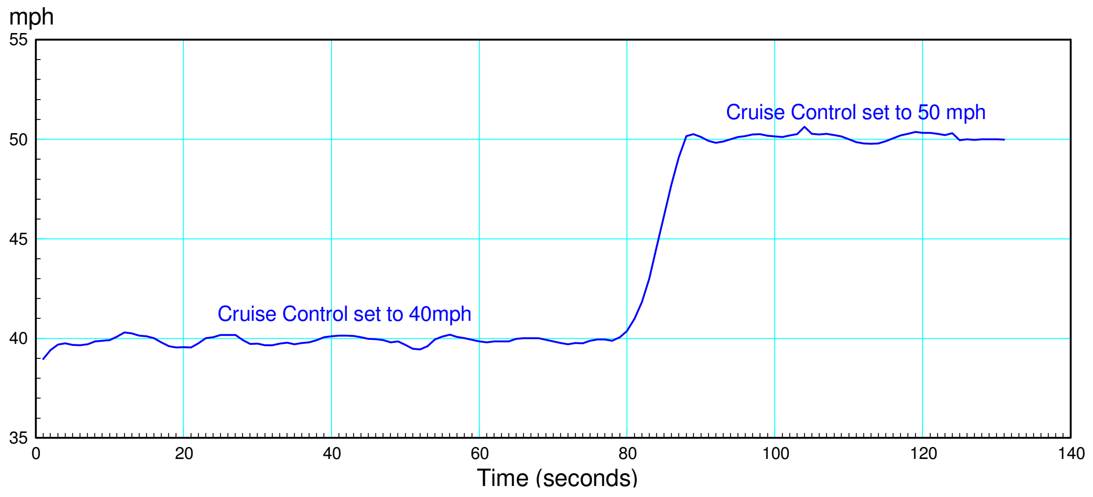
```
while(Button15.value() == 1):
    Error_Flag = 1
    while(Error_Flag == 1):
        Error_Flag = 0
        [t, x, y, v] = GPS_Read(0)
    if(Button14.value() == 0):
        Record_Flag = not Record_Flag
        if(Record_Flag):
            Beep()
            f = open(FileName, "a")
            print('Recording')
            LCD.Text('Recording',5,5,Pink,Navy)
        else:
            Beep()
            sleep(0.1)
            Beep()
            f.close()
            print('File Closed')
            LCD.Text('          ',5,5,Pink,Navy)
        while(Button14.value() == 0):
            pass

    Display(v*1.15078)

    if(Record_Flag):
        msg = str('{:9.4f}'.format(v*1.15077) + ' ' )
        f.write(msg + '\n')
```

Main Routine for GPS Speedometer



Resulting display of your speed in mph

In order to test the accuracy of the speedometer, you need some reference to test against. I don't have that. So, I placed the GPS speedometer in my car, found a long straight flat road, and set the cruise control to 40mph then 50mph. The results are as follows:



Measured speed with the cruise control set to 40mph and 50mph

There is variation in the readings. Between 10 and 70 seconds (when the cruise control was set to 40mph), the reading has

- Mean = 39.8913 mph
- St Dev = 0.1977 mph

Between 100 and 120 seconds (when the cruise control was set to 50mph), the readings have

- Mean = 50.0601 mph
- St Dev = 0.1954 mph

I can't tell whether this variation is due to the cruise control not keeping the speed absolutely steady or the GPS sensor's noise. Regardless, it's still a pretty accurate and easy to read speedometer.

Summary

GPS sensors are fairly inexpensive costing as little as $5 each from Amazon. With them you can determine where you are to within about 5 meters and your speed to within about 0.3 mph. Presumably, more expensive GPS sensors will work even better.

GPS sensors communicate with the Pi-Pico using SCI protocol. With some coding, the GPS messages can be pulled out and the fields can be read fairly easily. What you do with this is up to you and your creativity.

References

Pi-Pico and MicroPython

- https://github.com/geeekpi/pico_breakboard_kit
- https://micropython.org/download/RPI_PICO/
- https://learn.pimoroni.com/article/getting-started-with-pico
- https://www.w3schools.com/python/default.asp
- https://docs.micropython.org/en/latest/pyboard/tutorial/index.html

- https://docs.micropython.org/en/latest/library/index.html
- https://www.fredscave.com/02-about.html

Pi-Pico Breadboard Kit

- https://wiki.52pi.com/index.php?title=EP-0172

Other

- https://docs.sunfounder.com/projects/sensorkit-v2-pi/en/latest/
- https://electrocredible.com/raspberry-pi-pico-external-interrupts-button-micropython/
- https://peppe8o.com/adding-external-modules-to-micropython-with-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-esp32-esp8266-micropython/