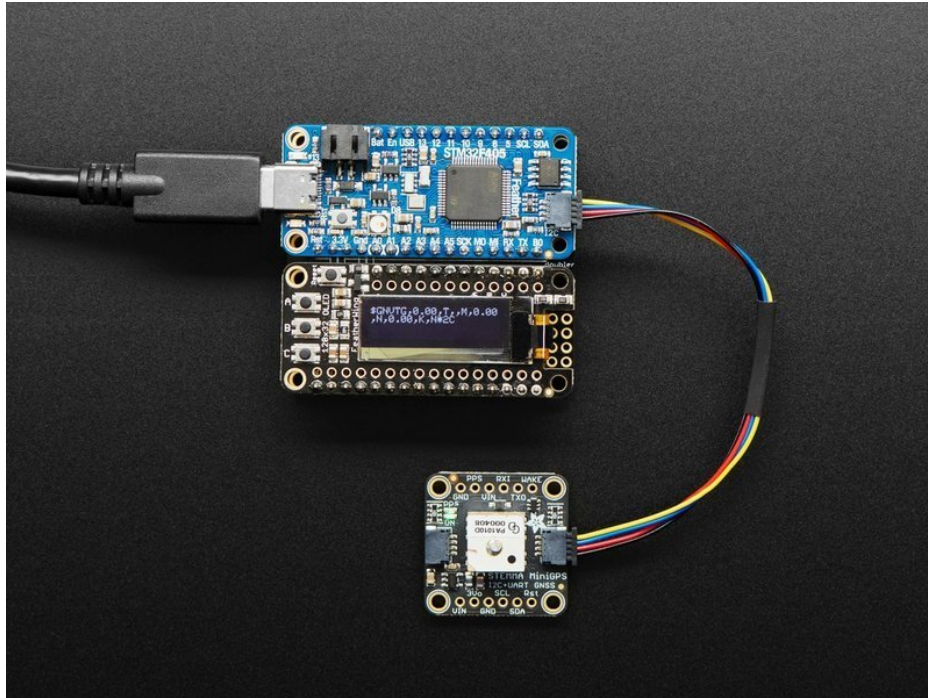
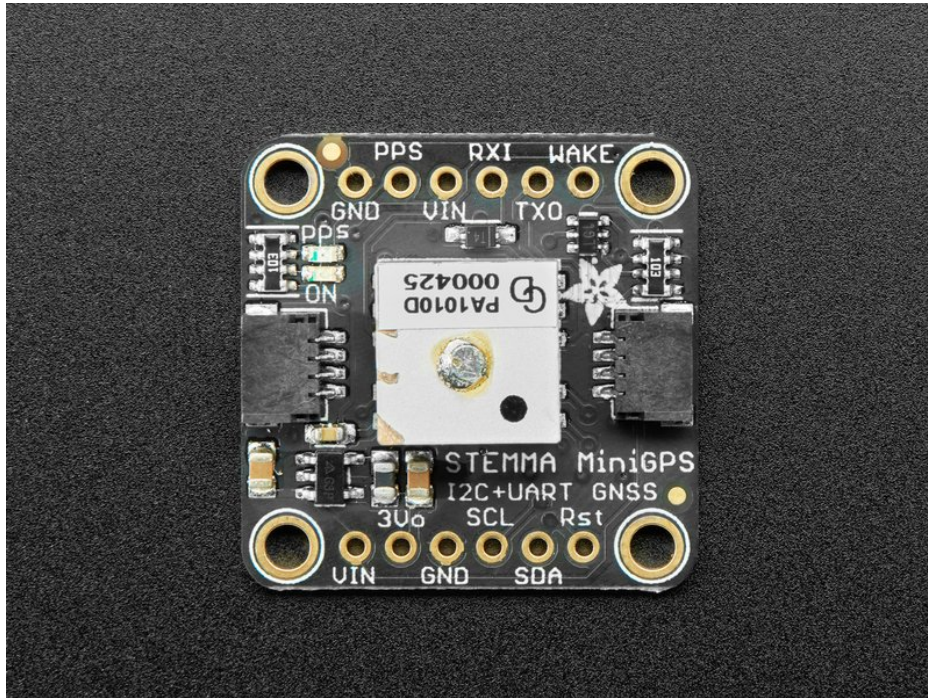


Overview

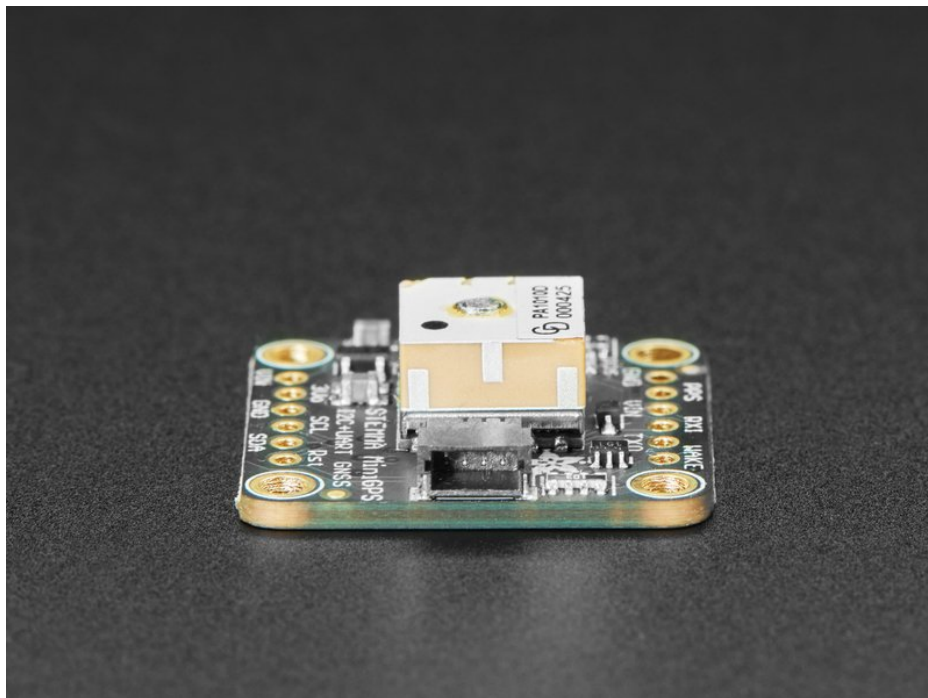


This miniature GPS breakout is only 1" x 1" (~ 25mm x 25mm) but houses a complete GPS/GNSS solution with both I2C and UART interfaces. There's even an antenna on top so it's plug and play!

- Support for GPS, GLONASS, GALILEO, QZSS
- -165 dBm sensitivity, up to 10 Hz updates
- Up to 210 PRN channels with 99 search channels and 33 simultaneous tracking channels
- 5V friendly design and only 30mA current draw
- Breadboard-able, with 4 mounting holes
- UART *and* I2C interfaces, pick whichever you like most!
- RTC battery-compatible
- PPS output on fix ± 20 ns jitter
- Internal patch antenna
- Low-power and standby mode with WAKE pin

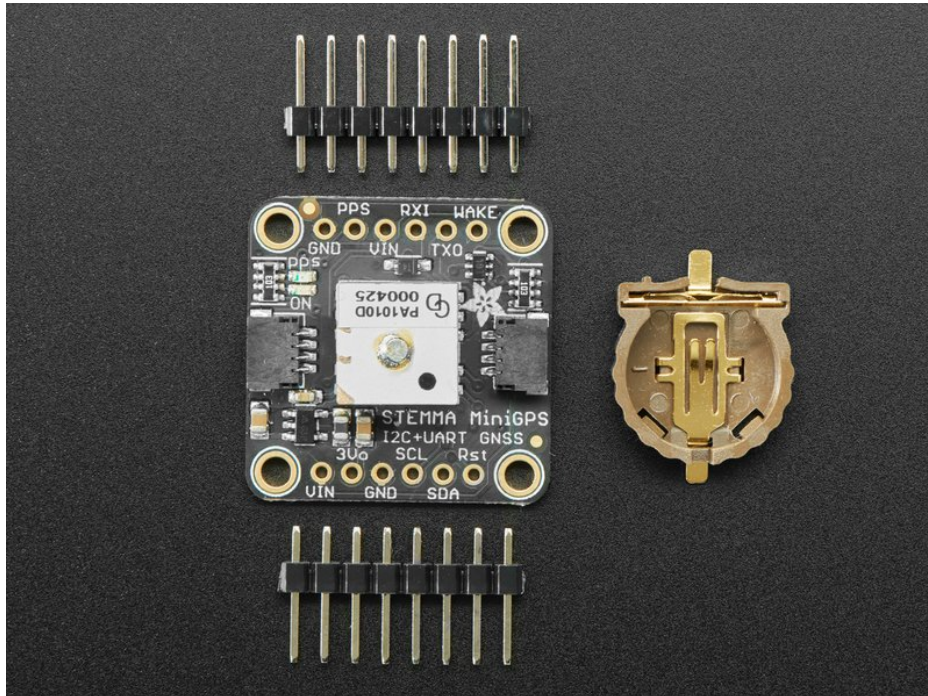


The breakout is built around the MTK3333 chipset, a reliable, high-quality GPS module that can handle up to 33 simultaneous tracking channels, has an excellent high-sensitivity receiver (-165 dBm tracking!), and a built in antenna. It can do up to 10 location updates a second for high speed, high sensitivity logging or tracking. Power usage is incredibly low, only 30 mA during navigation.



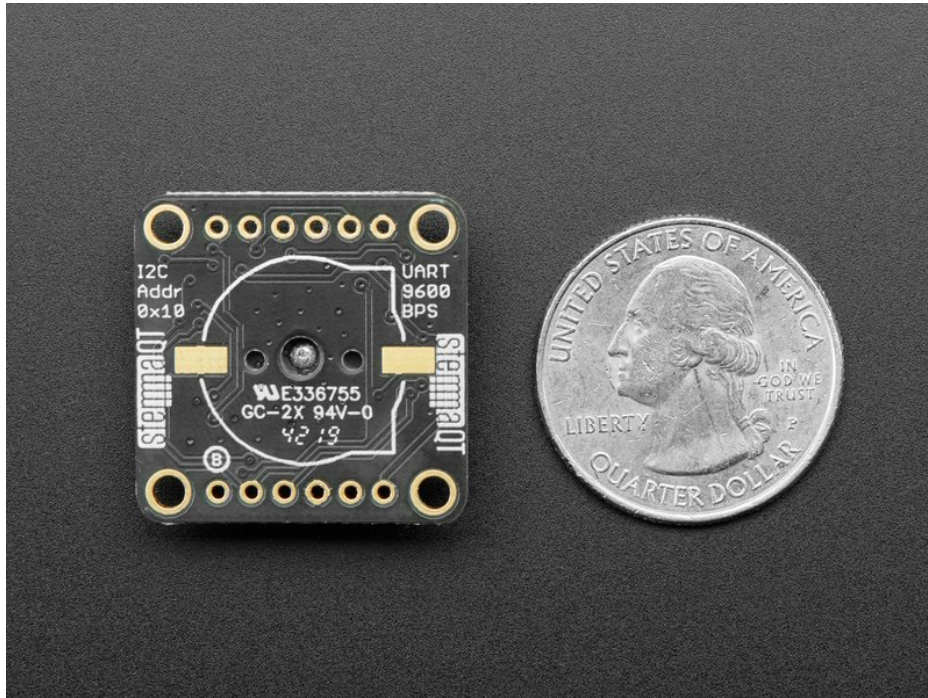
Best of all, we added all the extra goodies you could ever want: a ultra-low dropout 3.3V regulator so you can power it with 3.3-5VDC in, 5V level safe inputs on UART and I2C, a footprint for optional CR1220 coin cell to keep the RTC running and allow warm starts, a green power LED and a tiny red PPS LED. The LED blinks at about 1Hz when a fix is found and is off when no fix.

Unlike our Ultimate GPS modules, this module does not have the ability to connect an external antenna, it's designed to be as small as possible for compact projects.

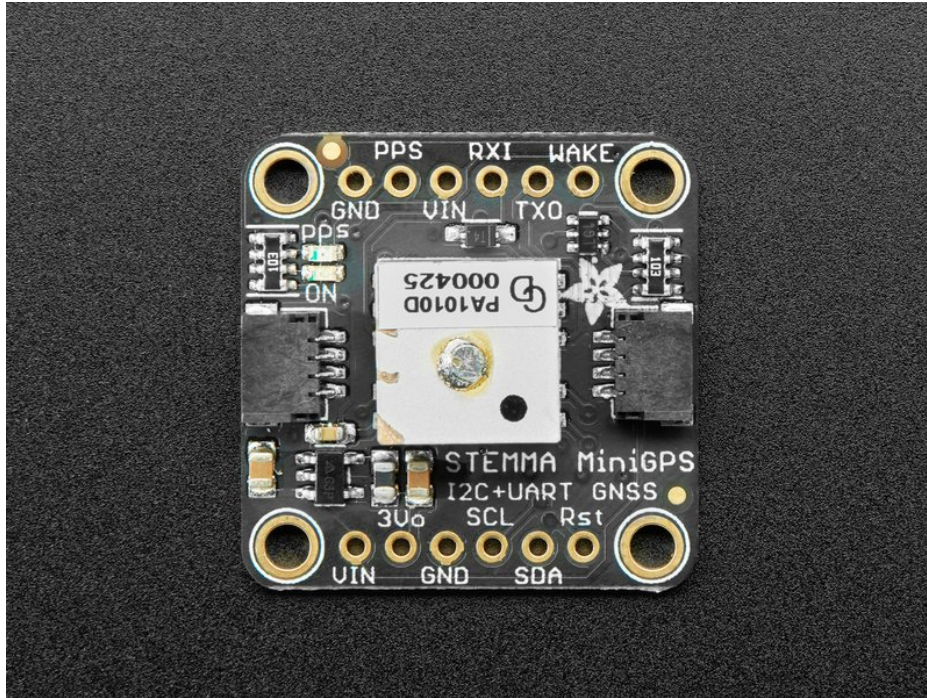


As with all Adafruit breakouts, we've done the work to make this GPS module super easy to use. We've put it on a breakout board with the required support circuitry and connectors to make it easy to work with, and is now a trend we've added SparkFun Qwiic (<https://adafru.it/Fpw>) compatible STEMMA QT (<https://adafru.it/Ft4>) JST SH connectors that allow you to get going **without needing to solder**. Just use a STEMMA QT adapter cable (<https://adafru.it/FA->), plug it into your favorite micro or Blinka supported SBC and you're ready to rock!

Comes with one fully assembled and tested module, a piece of header you can solder to it for bread-boarding, and a CR1220 coin cell holder. A CR1220 coin cell is not included, but we have them in the shop if you'd like to use the GPS's RTC (<http://adafru.it/380>)

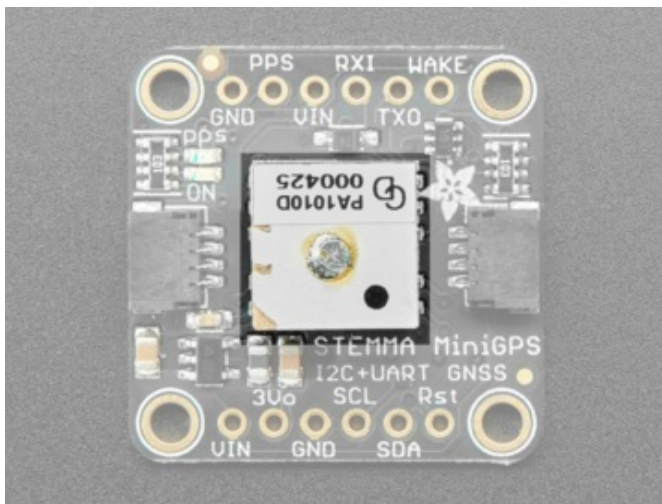


Pinouts



This GPS module can be used with I2C or UART. Let's take a look!

GPS Module

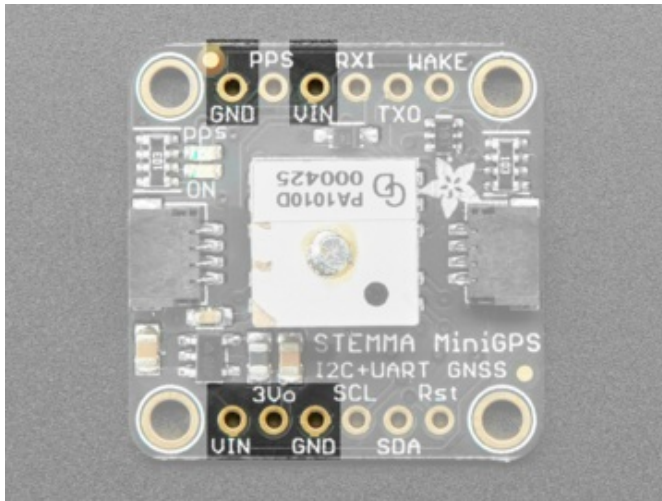


The **PA1010D GPS module with built-in antenna** is located in the center of the board. It has all kinds of features!

- Support for GPS, GLONASS, GALILEO, QZSS
- -165 dBm sensitivity, up to 10 Hz updates
- Up to 210 PRN channels with 99 search channels and 33 simultaneous tracking channels
- UART *and* I2C interfaces, pick whichever you like most!
- PPS output on fix ± 20 ns jitter
- Internal patch antenna
- Low-power and standby mode with WAKE pin

Note: Due to the sensitivity of the built in antenna, the PA1010D Mini GPS module may need a more unobstructed view of the sky than other GPS modules. If you are having trouble getting a fix, try moving the module to a clear spot with the antenna pointing up at the sky.

Power Pins

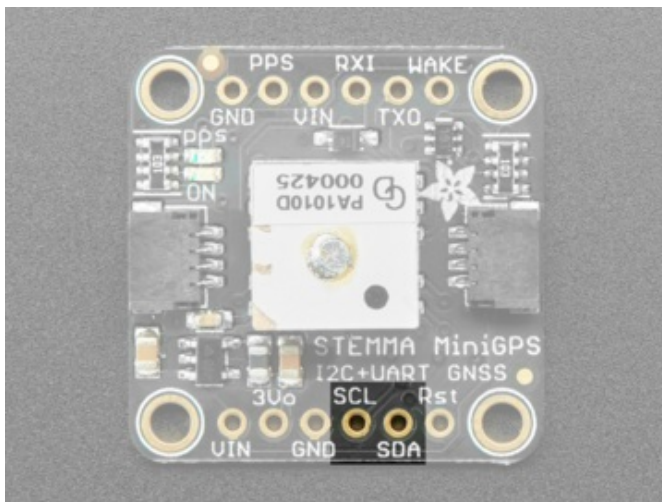


- **VIN** - power input, connect to 3-5VDC. It's important to connect to a *clean and quiet* power supply. GPS's are very sensitive, so you want a nice and quiet power supply. Don't connect to a switching supply if you can avoid it, an LDO will be less noisy! This module only draws 30mA current during navigation
- **GND** - power and signal ground. Connect to your power supply and microcontroller ground.

Optional:

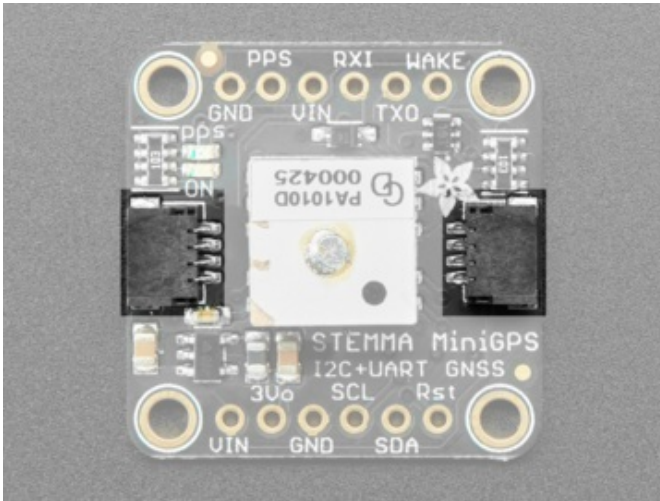
- **3Vo** - is the output from the onboard 3.3V regulator. If you have a need for a clean 3.3V output, you can use this! It can provide at least 100mA output.

I2C Data Pins



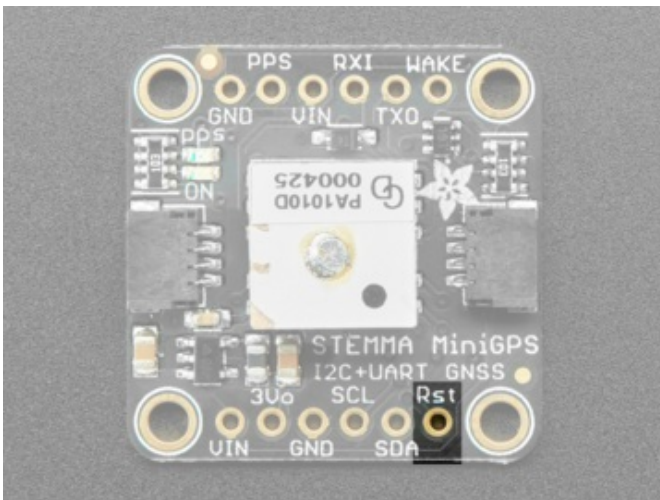
- **SCL** - this is the I2C clock pin, connect to your microcontroller or computer's I2C clock line.
- **SDA** - this is the I2C data pin, connect to your microcontroller or computer's I2C data line.

These pins have 10K pullups to Vin. They are level shifted so you can use 3 or 5V logic

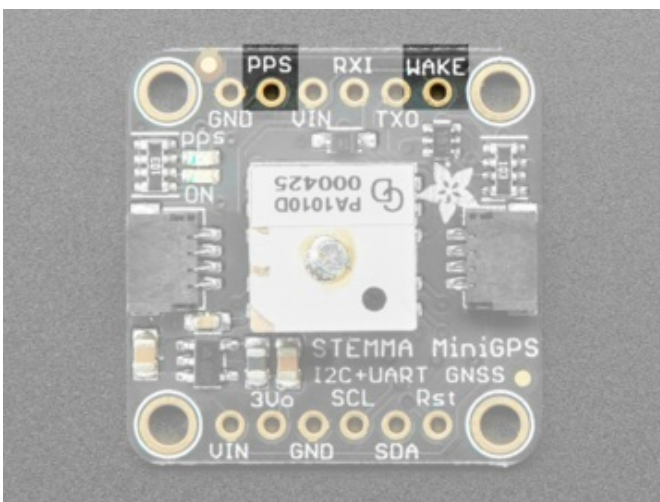


- On both sides in the middle are the [Sparkfun Qwiic](https://adafru.it/Fpw) (<https://adafru.it/Fpw>) compatible **STEMMA QT** (<https://adafru.it/Ft4>) JST SH connectors, for using with I2C. Use with any of the STEMMA QT cables available in the Adafruit shop to connect this breakout to your project without needing to solder!

Other Pins

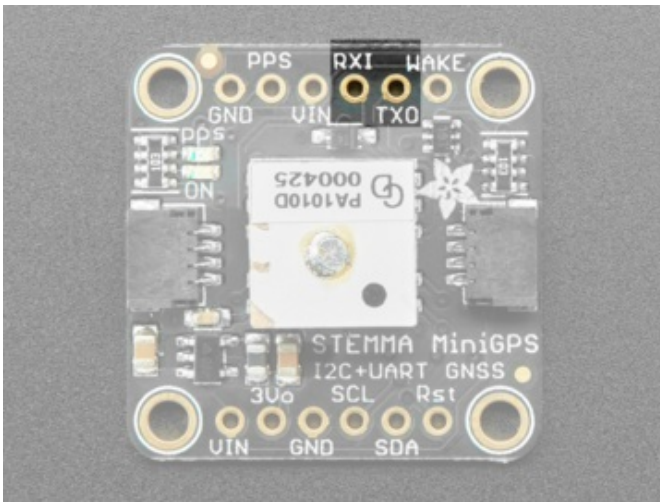


- **RST** - When pulled to ground, this will put the chip in the module into reset. Handy when you want to start with a completely clean setup.



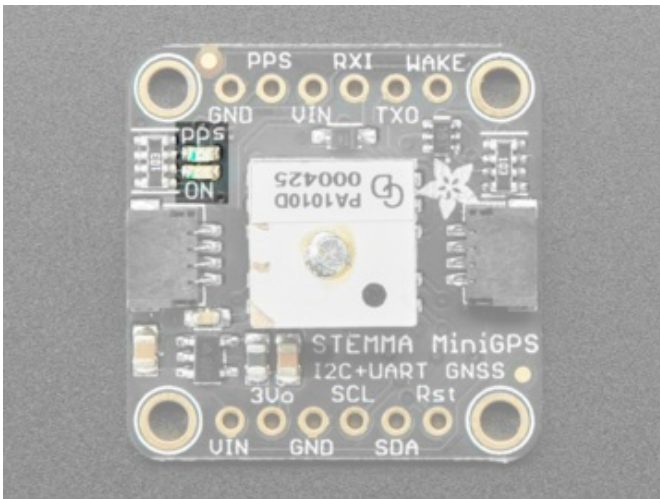
- **PPS** is a "pulse per second" output. Most of the time it is at logic low (ground) and then it pulses high (3.3V) once a second, for 50-100ms, so it should be easy for a microcontroller to sync up to it
- **WAKE** - This pin works with low power and standby modes. Check the datasheet for more information!

UART Serial Data Pins



- **TXO** - the pin that transmits data *from* the GPS module to your microcontroller or computer. It is 3.3V logic level. Data comes out at 9600 baud 8N1 by default
- **RXI** - the pin that you can use to send data *to* the GPS. This pin is level shifted so you can use 3-5V logic. By default it expects 9600 baud data by default.

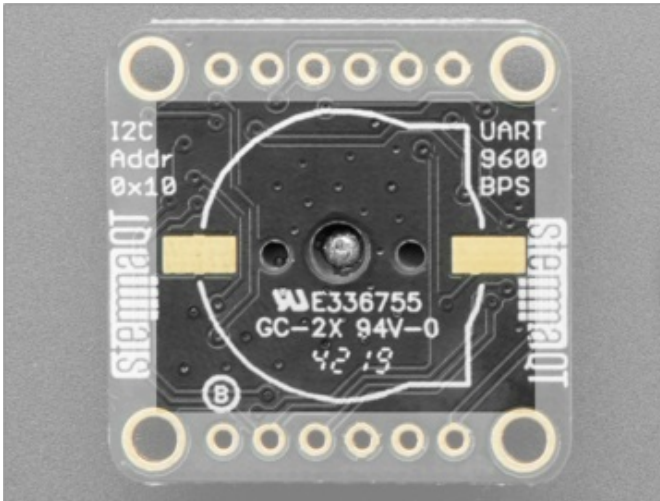
LEDs



There are two LEDs on the board.

- **ON** - Green power LED. Lit when the board is powered
- **PPS** - Red PPS LED, blinks at about 1Hz when a fix is found and is off when no fix.

Optional Coin Cell

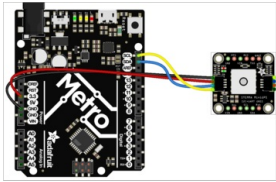


The back has a footprint for an **optional coin cell battery**. The board ships with a CR1220 coin cell holder that can be soldered onto the back. CR1220 battery not included.

Arduino I2C Usage

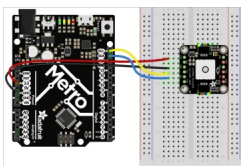
Wiring

If you have a [STEMMA QT breakout cable](https://adafru.it/FA-) (<https://adafru.it/FA->), then you can wire like this:



- RED to 3.3
- BLACK to GND
- BLUE to SDA
- YELLOW to SCL

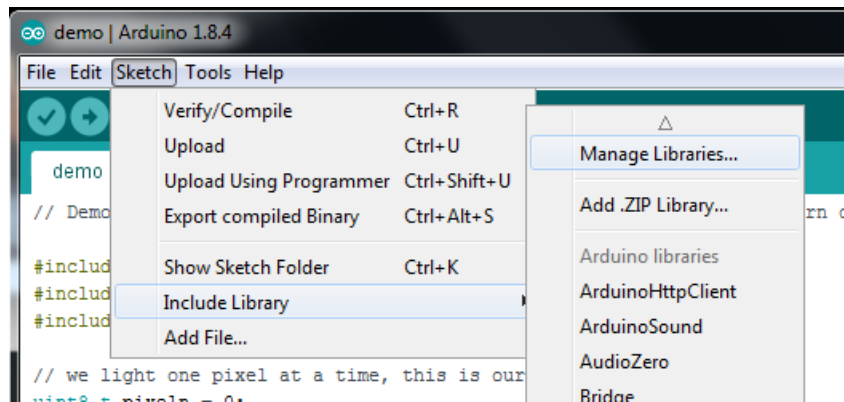
If you want to solder pins to the header connectors and use a breadboard, then wire like this:



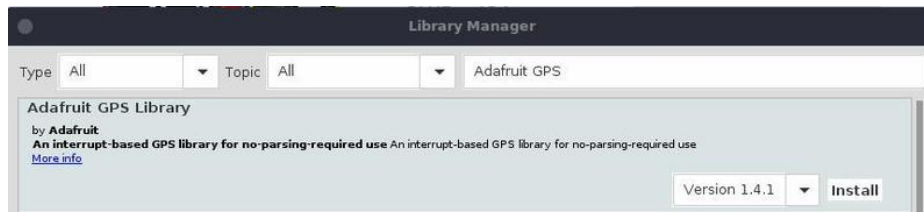
- VIN to 3.3
- GND to GND
- SDA to SDA
- SCL to SCL

Installation

The **Adafruit GPS Library** includes support for the Mini GPS PA1010D module. You can install it from the Arduino IDE via the Library Manager:



Click the **Manage Libraries...** menu item, search for **Adafruit GPS**, and select the **Adafruit GPS Library**:



Echo Test Demo

We can test basic functionality using one of the examples from the library. This won't do any parsing. It simply dumps the raw data sentences as received from the GPS module.

Open up **File -> Examples -> Adafruit GPS Library -> GPS_I2C_EchoTest** and upload to your Arduino board with the GPS module connected.

Once the sketch is uploaded, open up the Serial Monitor (**Tools -> Serial Monitor**) at 115200 baud. You should see something like this:

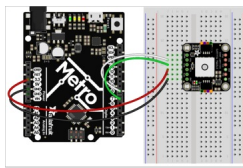


```
Adafruit GPS library basic I2C test!  
$GNGGA,235945.100,,,,,0,0,,M,,*5B  
$GPGSA,A,1,,,,,,,,,,,,,*1E  
$GLGSA,A,1,,,,,,,,,,,,,*02  
$GNRMC,235945.100,V,,,,,0.00,0.00,050180,,N*52  
$GNVTG,0.00,T,,M,0.00,N,0.00,K,N*2C  
$GNGGA,235946.099,,,,,0,0,,M,,*59  
$GPGSA,A,1,,,,,,,,,,,,,*1E  
$GLGSA,A,1,,,,,,,,,,,,,*02  
$GNRMC,235946.099,V,,,,,0.00,0.00,050180,,N*50  
$GNVTG,0.00,T,,M,0.00,N,0.00,K,N*2C  
$GNGGA,235947.099,,,,,0,0,,M,,*58  
$GPGSA,A,1,,,,,,,,,,,,,*1E  
$GLGSA,A,1,,,,,,,,,,,,,*02  
$GPGSV,1,1,00*79  
$GLGSV,1,1,00*65
```

If you see this, then you are successfully talking to the GPS module. Once it obtains a lock on the GPS satellites, which can take several minutes, there will be more information in the sentences. To see that information in a more user friendly format, try running the **GPS_I2C_Parsing** example from the library.

Wiring

Note that the breakout has pins on two sides. Be sure to use the side with the UART pins. They are labeled **TXO** and **RXI**. We'll demonstrate using **SoftwareSerial** on the Metro 328. Here's the wiring:



- 3.3 to VIN
- GND to GND
- 8 to TXO
- 7 to RXI

Installation

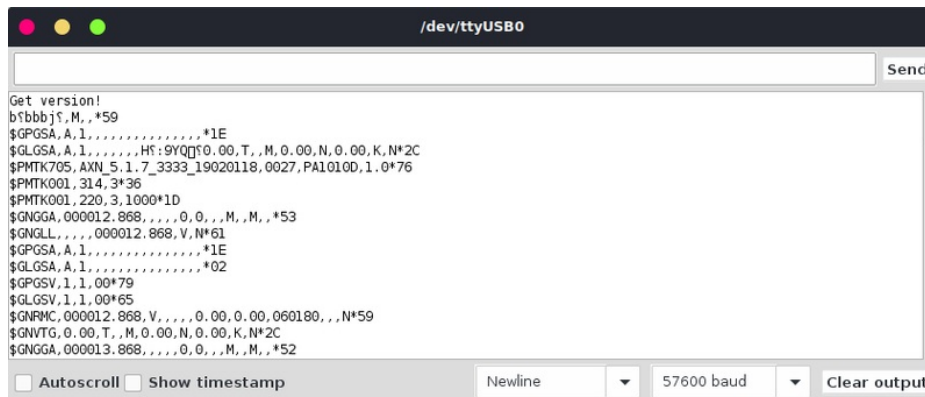
See the Arduino I2C Usage section for details about installing the **Adafruit GPS Library**. The same library is used for UART.

Echo Test Demo

We can test basic functionality using one of the examples from the library. This won't do any parsing. It simply dumps the raw data sentences as received from the GPS module.

Open up **File -> Examples -> Adafruit GPS Library -> GPS_SoftwareSerial_EchoTest** and upload to your Arduino board with the GPS module connected.

Once the sketch is uploaded, open up the Serial Monitor (**Tools -> Serial Monitor**) at 115200 baud. You should see something like this:



If you see this, then you are successfully talking to the GPS module. Once it obtains a lock on the GPS satellites, which can take several minutes, there will be more information in the sentences. To see that information in a more user friendly format, try running the **GPS_SoftwareSerial_Parsing** example from the library.

Hardware UART

The above example demonstrated UART usage via **SoftwareSerial**. If you have a board with an available hardware UART, you can use that also. Simply connect to the TX/RX pins for the hardware UART for your particular board and then see the examples in the **Adafruit GPS Library** with **HardwareSerial** in the name.

It's easy to use the Adafruit Mini GPS PA1010D breakout with Python or CircuitPython and the [Adafruit CircuitPython GPS](https://adafru.it/BuR) (<https://adafru.it/BuR>) module. This library allows you to write Python code that reads the date, time, location and more from the breakout.

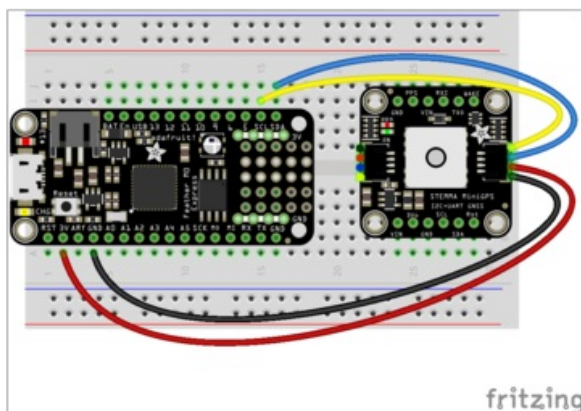
You can use this sensor with any CircuitPython microcontroller board or with a computer that has GPIO and Python thanks to [Adafruit_Blinka](https://adafru.it/BSN), our [CircuitPython-for-Python compatibility library](https://adafru.it/BSN) (<https://adafru.it/BSN>).

CircuitPython Microcontroller Wiring

The Adafruit Mini GPS PA1010D can be wired up in multiple ways. We recommend I2C as it is the simplest. There are two ways you can connect the GPS to a microcontroller using I2C.

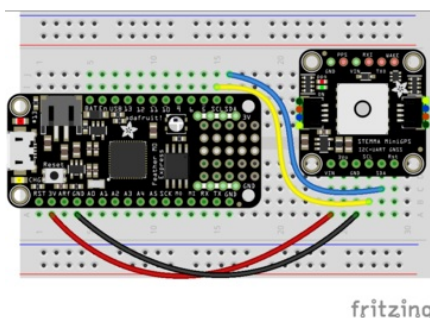
I2C Interface

Here is an example of the module connected to a Feather M0 Express for I2C using the STEMMA connector and a STEMMA cable:



- Feather 3V to STEMMA red wire (VIN)
- Feather GND to STEMMA black wire (GND)
- Feather SDA to STEMMA blue wire (SDA)
- Feather SCL to STEMMA yellow wire (SCL)

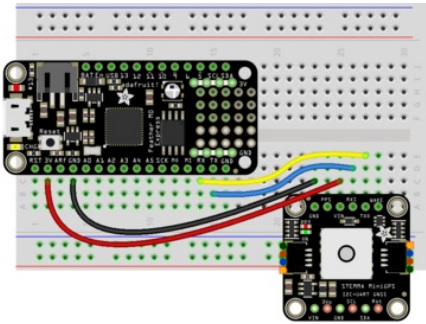
Here is an example of the module connected to a Feather M0 Express for I2C using jumper wires:



- Feather 3V to module VIN
- Feather GND to module GND
- Feather SCL to module SCL
- Feather SDA to module SDA

UART Interface

Here is an example of the module connected to a Feather M0 Express using UART:



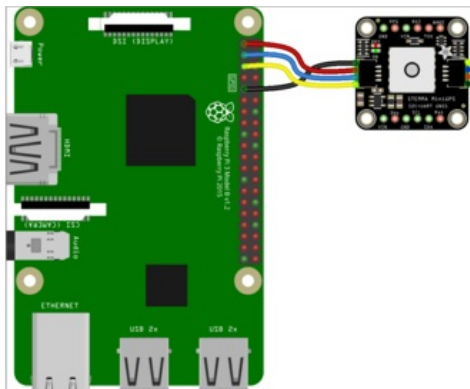
- Feather 3V to module VIN
- Feather GND to module GND
- Feather TX to module RXI
- Feather RX to module TXO

Python Computer Wiring

Since there's *dozens* of Linux computers/boards you can use we will show wiring for Raspberry Pi. For other platforms, please visit the [guide for CircuitPython on Linux](https://adafru.it/BSN) to see whether your platform is supported (<https://adafru.it/BSN>).

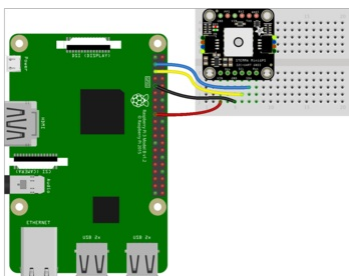
I2C Interface

Here's the Raspberry Pi wired with I2C using the STEMMA connector and a STEMMA cable:



- Pi 3V to STEMMA red wire (VIN)
- Pi GND to STEMMA black wire (GND)
- Pi SDA to STEMMA blue wire (SDA)
- Pi SCL to STEMMA yellow wire (SCL)

Here's the Raspberry Pi wired with I2C using jumper wires:



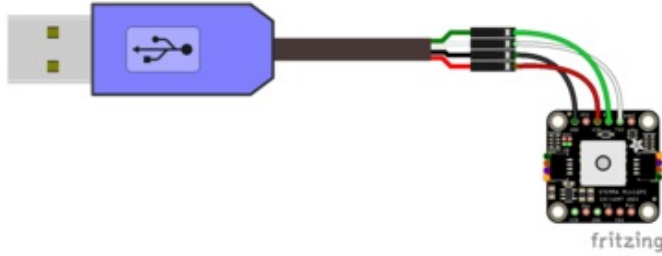
- Pi 3V to module VIN
- Pi GND to module GND
- Pi SCL to module SCL
- Pi SDA to module SDA

UART Interface

For UART, you have two options: An external USB-to-serial converter or the built-in UART on the Pi's TX/RX pins.

USB-to-Serial Cable Interface

Here's an example of wiring up the [USB-to-TTL serial converter](https://adafru.it/dDd) (<https://adafru.it/dDd>), and the [FTDI serial TTL-232 USB cable](https://adafru.it/dNN) (<https://adafru.it/dNN>) (also available in [USB-C](https://adafru.it/H3d) (<https://adafru.it/H3d>)):

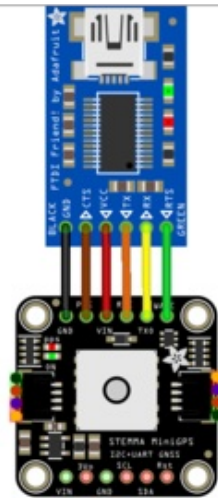


For **USB to TTL serial cable**:

- **USB 3V (red wire) to module VIN**
- **USB GND (black wire) to module GND**
- **USB TX (green wire) to module RXI**
- **USB RX (white wire) to module TXO**

For **FTDI serial TTL cable** - the FTDI cable pinout matches the pinout on the UART side of the PA1010D Mini GPS Module. Connect the cable so that the wires align as follows:

- **FTDI black wire to module GND**
- **FTDI brown wire to module PPS**
- **FTDI red wire to module VIN**
- **FTDI orange wire to module RXI**
- **FTDI yellow wire to module TXO**
- **FTDI green wire to module WAKE**

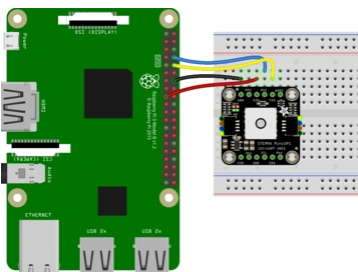


Hardware UART Interface



For single board computers other than the Raspberry Pi, the serial port may be tied to the console or not be available to the user. Please see the board documentation to see how the serial port may be used.

Here's an example using the Pi's built-in UART:



- **Pi 3V to module VIN**
- **Pi GND to module GND**
- **Pi TX to module RXI**
- **Pi RX to module TXO**

If you want to use the built-in UART, you'll need to disable the serial console and enable the serial port hardware in **raspi-config**. See the **UART/Serial** section of the **CircuitPython on Raspberry Pi guide** (<https://adafru.it/CEk>) for detailed

instructions on how to do this.

CircuitPython Installation of GPS Library

Next you'll need to install the [Adafruit CircuitPython GPS \(https://adafru.it/BuR\)](https://adafru.it/BuR) library on your CircuitPython board.

First make sure you are running the [latest version of Adafruit CircuitPython \(https://adafru.it/Em8\)](https://adafru.it/Em8) for your board.

Next you'll need to install the necessary libraries to use the hardware. Carefully follow the steps to find and install these libraries from [Adafruit's CircuitPython library bundle \(https://adafru.it/ENC\)](https://adafru.it/ENC). For example, the Welcome to CircuitPython guide has [a great page on how to install the library bundle \(https://adafru.it/ABU\)](https://adafru.it/ABU).

To install the libraries, you'll need to copy the following files from the bundle to the **lib** folder on your **CIRCUITPY** drive:

- `adafruit_gps.mpy`
- `adafruit_bus_device`

Before continuing make sure your board's **lib** folder has the `adafruit_gps.mpy` and `adafruit_bus_device` files and folders copied over.

Python Installation of GPS Library

You'll need to install the `Adafruit_Blinka` library that provides the CircuitPython support in Python. This may also require enabling I2C and UART on your platform and verifying you are running Python 3. [Since each platform is a little different, and Linux changes often, please visit the CircuitPython on Linux guide to get your computer ready \(https://adafru.it/BSN\)!](https://adafru.it/BSN)

Once that's done, from your command line run the following command:

- `sudo pip3 install adafruit-circuitpython-gps`


If your default Python is version 3 you may need to run 'pip' instead. Just make sure you aren't trying to use CircuitPython on Python 2.x, it isn't supported!

Python Docs

Python Docs (<https://adafru.it/C4M>)

CircuitPython & Python I2C Usage

To demonstrate the usage of the GPS module in CircuitPython using I2C, let's look at a complete program example, the `gps_echotest.py` file from the module's examples.

 To use this example with I2C, you must make some changes to the code.

To use this example with I2C, you need to change four separate lines of code found towards the beginning of the initialisation section of the example.

First, comment out the following lines by adding a `#` to the beginning of each line:

```
uart = busio.UART(board.TX, board.RX, baudrate=9600, timeout=10)

gps = adafruit_gps.GPS(uart, debug=False)
```

Then, uncomment the following lines by removing the `#` from the beginning of each line:

```
#i2c = busio.I2C(board.SCL, board.SDA)

#gps = adafruit_gps.GPS_GtopI2C(i2c, debug=False) # Use I2C interface
```

Once these changes are made, you are ready to continue.

CircuitPython Microcontroller Usage

With a CircuitPython microcontroller, save the `gps_echotest.py` file as `code.py` on your **CIRCUITPY** drive. Then connect to the serial console to see the output.

Linux, Computer or Raspberry Pi Usage

From the command line, run the following command:

```
python3 gps_echotest.py
```

Echotest Example

```
# Simple GPS module demonstration.
# Will print NMEA sentences received from the GPS, great for testing connection
# Uses the GPS to send some commands, then reads directly from the GPS
import time
import board
import busio

import adafruit_gps

# Create a serial connection for the GPS connection using default speed and
```

```

# a slightly higher timeout (GPS modules typically update once a second).
# These are the defaults you should use for the GPS FeatherWing.
# For other boards set RX = GPS module TX, and TX = GPS module RX pins.
uart = busio.UART(board.TX, board.RX, baudrate=9600, timeout=10)

# for a computer, use the pyserial library for uart access
# import serial
# uart = serial.Serial("/dev/ttyUSB0", baudrate=9600, timeout=10)

# If using I2C, we'll create an I2C interface to talk to using default pins
# i2c = busio.I2C(board.SCL, board.SDA)

# Create a GPS module instance.
gps = adafruit_gps.GPS(uart) # Use UART/pyserial
# gps = adafruit_gps.GPS_GtopI2C(i2c) # Use I2C interface

# Initialize the GPS module by changing what data it sends and at what rate.
# These are NMEA extensions for PMTK_314_SET_NMEA_OUTPUT and
# PMTK_220_SET_NMEA_UPDATERATE but you can send anything from here to adjust
# the GPS module behavior:
# https://cdn-shop.adafruit.com/datasheets/PMTK_A11.pdf

# Turn on the basic GGA and RMC info (what you typically want)
gps.send_command(b'PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0')
# Turn on just minimum info (RMC only, location):
# gps.send_command(b'PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
# Turn off everything:
# gps.send_command(b'PMTK314,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
# Turn on everything (not all of it is parsed!)
# gps.send_command(b'PMTK314,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0')

# Set update rate to once a second (1hz) which is what you typically want.
gps.send_command(b'PMTK220,1000')
# Or decrease to once every two seconds by doubling the millisecond value.
# Be sure to also increase your UART timeout above!
# gps.send_command(b'PMTK220,2000')
# You can also speed up the rate, but don't go too fast or else you can lose
# data during parsing. This would be twice a second (2hz, 500ms delay):
# gps.send_command(b'PMTK220,500')

# Main loop runs forever printing data as it comes in
timestamp = time.monotonic()
while True:
    data = gps.read(32) # read up to 32 bytes
    # print(data) # this is a bytearray type

    if data is not None:
        # convert bytearray to string
        data_string = "".join([chr(b) for b in data])
        print(data_string, end="")

    if time.monotonic() - timestamp > 5:
        # every 5 seconds...
        gps.send_command(b'PMTK605') # request firmware version
        timestamp = time.monotonic()

```

Connect to the serial console. You should see something like the following output.

```
$GNGGA,003837.101,,,,,0,0,,M,M,,*59
$GNRMC,003837.101,V,,,,,0.00,0.00,060180,,N*53
$GNGGA,003838.101,,,,,0,0,,M,M,,*56
$GNRMC,003838.101,V,,,,,0.00,0.00,060180,,N*5C
$GNGGA,003839.101,,,,,0,0,,M,M,,*57
$GNRMC,003839.101,V,,,,,0.00,0.00,060180,,N*5D
$GNGGA,003840.101,,,,,0,0,,M,M,,*59
$GNRMC,003840.101,V,,,,,0.00,0.00,060180,,N*53
$GNGGA,003841.101,,,,,0,0,,M,M,,*58
$GNRMC,003841.101,V,,,,,0.00,0.00,060180,,N*52
$GNGGA,003830.101,,,,,0,0,,M,M,,*5E
$GNRMC,003830.101,V,,,,,0.00,0.00,060180,,N*54
$GNGGA,003831.101,,,,,0,0,,M,M,,*5F
$GNRMC,003831.101,V,,,,,0.00,0.00,060180,,N*55
$PMTK705,AXN_5.1.7_3333_19020118,0027,PA1010D,1.0*76
```

This is the raw GPS "NMEA sentence" output from the module. There are a few different kinds of NMEA sentences, the most common ones people use are the **\$GPRMC** (Global Positioning Recommended Minimum Coordinates or something like that) and the **\$GPGGA** sentences. These two provide the time, date, latitude, longitude, altitude, estimated land speed, and fix type. Fix type indicates whether the GPS has locked onto the satellite data and received enough data to determine the location (2D fix) or location+altitude (3D fix).

For more details about NMEA sentences and what data they contain, check out this site (<https://adafru.it/kMb>)

If you look at the data in the above window, you can see that there are a lot of commas, with no data in between them. That's because this module is on my desk, indoors, and does not have a 'fix'. To get a fix, we need to put the module outside.

Note: Due to the antenna being built in, the PA1010D Mini GPS module may need a more unobstructed view of the sky than other GPS modules with external antennae. If you are having trouble getting a fix, try moving the module to a more ideal location.

GPS modules will always send data EVEN IF THEY DO NOT HAVE A FIX! In order to get 'valid' (not-blank) data you must have the GPS module directly outside, with the square GPS module pointing up with a clear sky view. In ideal conditions, the module can get a fix in under 45 seconds. however depending on your location, satellite configuration, solar flares, tall buildings nearby, RF noise, etc it may take up to half an hour (or more) to get a fix! This does not mean your GPS module is broken, the GPS module will always work as fast as it can to get a fix.

For an explanation of the rest of the setup in this example, see the [GPS Example Code Explained](https://adafru.it/H3E) section of the [CircuitPython & Python UART Usage](https://adafru.it/H3E) page (<https://adafru.it/H3E>).

Following setup is the main loop.

```
while True:
    data = gps.read(32) # read up to 32 bytes
    # print(data) # this is a bytearray type

    if data is not None:
        # convert bytearray to string
        data_string = ''.join([chr(b) for b in data])
        print(data_string, end="")

    if time.monotonic() - timestamp > 5:
        # every 5 seconds...
        gps.send_command(b'PMTK605') # request firmware version
        timestamp = time.monotonic()
```

First we read up to 32 bytes directly from the GPS and save it to `data` as a bytearray.

Next, we check to see that data has been read by verifying that it is not equal to `None` - this avoids the code failing when no data is returned. Then we convert the bytearray into a string and print it out.

Lastly, every 5 seconds, we request the firmware version.

Once you've used the Echotest to verify that your Mini GPS module is connected and working, you can switch to the Simplest example to get a more readable version of the data. **To use the `gps_simplest.py` example with I2C, you must make the same changes shown above that you made to the Echotest example.** Comment out the `UART` setup lines and uncomment the `I2C` setup lines. Once the changes are made, you can follow along with the code explanation found in the [CircuitPython & Python UART Usage: Example Parsing Code](https://adafru.it/H3E) section (<https://adafru.it/H3E>).

CircuitPython & Python UART Usage

To demonstrate the usage of the GPS module in CircuitPython using UART, let's look at a complete program example, the `gps_simpletest.py` file from the module's examples.

CircuitPython Microcontroller

With a CircuitPython microcontroller, save this file as `code.py` on your board, then open the serial console to see its output.

Linux/Computer/Raspberry Pi with Python



If you're running `gps_simpletest.py` on the Raspberry Pi (or any computer), you'll have to make some changes.

On the Raspberry Pi, comment out the `uart = busio(...)` line, and uncomment the `import serial` and `uart = serial.Serial(...)` lines, changing `/dev/ttyUSB0` to the appropriate serial port. Now you can run the program with the following command:

```
python3 gps_simpletest.py
```

Example Parsing Code

```
# Simple GPS module demonstration.
# Will wait for a fix and print a message every second with the current location
# and other details.
import time
import board
import busio

import adafruit_gps

# Create a serial connection for the GPS connection using default speed and
# a slightly higher timeout (GPS modules typically update once a second).
# These are the defaults you should use for the GPS FeatherWing.
# For other boards set RX = GPS module TX, and TX = GPS module RX pins.
uart = busio.UART(board.TX, board.RX, baudrate=9600, timeout=10)

# for a computer, use the pyserial library for uart access
# import serial
# uart = serial.Serial("/dev/ttyUSB0", baudrate=9600, timeout=10)

# If using I2C, we'll create an I2C interface to talk to using default pins
# i2c = busio.I2C(board.SCL, board.SDA)

# Create a GPS module instance.
gps = adafruit_gps.GPS(uart, debug=False) # Use UART/pyserial
# gps = adafruit_gps.GPS_GtopI2C(i2c, debug=False) # Use I2C interface

# Initialize the GPS module by changing what data it sends and at what rate.
# These are NMEA extensions for PMTK_314_SET_NMEA_OUTPUT and
# PMTK_220_SET_NMEA_UPDATE but you can send anything from here to adjust
```



```

# PMTK_220_SET_UPDATE_RATE but you can send anything from here to adjust
# the GPS module behavior:
# https://cdn-shop.adafruit.com/datasheets/PMTK\_A11.pdf

# Turn on the basic GGA and RMC info (what you typically want)
gps.send_command(b'PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
# Turn on just minimum info (RMC only, location):
# gps.send_command(b'PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
# Turn off everything:
# gps.send_command(b'PMTK314,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
# Turn on everything (not all of it is parsed!)
# gps.send_command(b'PMTK314,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')


# Set update rate to once a second (1hz) which is what you typically want.
gps.send_command(b'PMTK220,1000')
# Or decrease to once every two seconds by doubling the millisecond value.
# Be sure to also increase your UART timeout above!
# gps.send_command(b'PMTK220,2000')
# You can also speed up the rate, but don't go too fast or else you can lose
# data during parsing. This would be twice a second (2hz, 500ms delay):
# gps.send_command(b'PMTK220,500')

# Main loop runs forever printing the location, etc. every second.
last_print = time.monotonic()
while True:
    # Make sure to call gps.update() every loop iteration and at least twice
    # as fast as data comes from the GPS unit (usually every second).
    # This returns a bool that's true if it parsed new data (you can ignore it
    # though if you don't care and instead look at the has_fix property).
    gps.update()
    # Every second print out current location details if there's a fix.
    current = time.monotonic()
    if current - last_print >= 1.0:
        last_print = current
        if not gps.has_fix:
            # Try again if we don't have a fix yet.
            print("Waiting for fix...")
            continue
        # We have a fix! (gps.has_fix is true)
        # Print out details about the fix like location, date, etc.
        print("=" * 40) # Print a separator line.
        print(
            "Fix timestamp: {}/{}/{ } {:02}:{:02}:{:02}".format(
                gps.timestamp_utc.tm_mon, # Grab parts of the time from the
                gps.timestamp_utc.tm_mday, # struct_time object that holds
                gps.timestamp_utc.tm_year, # the fix time. Note you might
                gps.timestamp_utc.tm_hour, # not get all data like year, day,
                gps.timestamp_utc.tm_min, # month!
                gps.timestamp_utc.tm_sec,
            )
        )
        print("Latitude: {0:.6f} degrees".format(gps.latitude))
        print("Longitude: {0:.6f} degrees".format(gps.longitude))
        print("Fix quality: {}".format(gps.fix_quality))
        # Some attributes beyond latitude, longitude and timestamp are optional
        # and might not be present. Check if they're None before trying to use!
        if gps.satellites is not None:
            print("# satellites: {}".format(gps.satellites))
        if gps.altitude_m is not None:
            print("Altitude: {} meters".format(gps.altitude_m))

```

```
if gps.speed_knots is not None:
    print("Speed: {}".format(gps.speed_knots))
if gps.track_angle_deg is not None:
    print("Track angle: {}".format(gps.track_angle_deg))
if gps.horizontal_dilution is not None:
    print("Horizontal dilution: {}".format(gps.horizontal_dilution))
if gps.height_geoid is not None:
    print("Height geo ID: {}".format(gps.height_geoid))
```

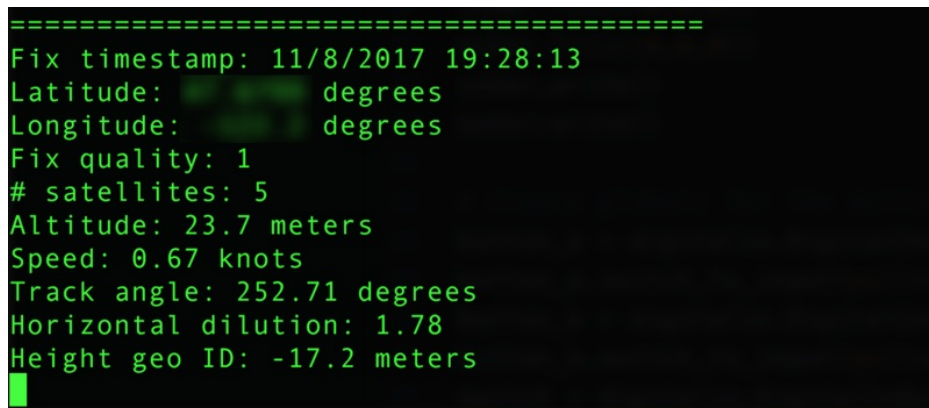
When the code runs it will print a message every second, either an update that it's still waiting for a GPS fix:



```
Waiting for fix...
Waiting for fix...
Waiting for fix...
Waiting for fix...
Waiting for fix...
Waiting for fix...
Waiting for fix...
Waiting for fix...
Waiting for fix...
```

Note: Due to the antenna being built in, the PA1010D Mini GPS module may need a more unobstructed view of the sky than other GPS modules with external antennae. With any GPS module, if you are having trouble getting a fix, try moving it to a more ideal location.

Once a fix has been established, it will print details about the current location and other GPS data:



```
=====
Fix timestamp: 11/8/2017 19:28:13
Latitude: 42.3456 degrees
Longitude: -71.1234 degrees
Fix quality: 1
# satellites: 5
Altitude: 23.7 meters
Speed: 0.67 knots
Track angle: 252.71 degrees
Horizontal dilution: 1.78
Height geo ID: -17.2 meters
█
```

Let's look at the code in a bit more detail to understand how it works. First the example needs to import a few modules like the built-in `busio` and `board` modules that access serial ports and other hardware:

```
import board
import busio
import time
```

Next the GPS module is imported:

```
import adafruit_gps
```

Now a `serial UART` (<https://adafru.it/BuT>) is created and connected to the serial port pins the GPS module will use, this

You don't need to worry about adding a NMEA checksum to your command either, the function will do this automatically (or not, set `add_checksum=False` as a parameter and it will skip the checksum addition).

Now we can jump into a main loop that continually updates data from the GPS module and prints out status. The most important part of this loop is calling the GPS update function:

```
# Make sure to call gps.update() every loop iteration and at least twice
# as fast as data comes from the GPS unit (usually every second).
# This returns a bool that's true if it parsed new data (you can ignore it
# though if you don't care and instead look at the has_fix property).
gps.update()
```

Like the comments mention, you must call `update` every loop iteration and ideally multiple times a second. Each time you call `update`, it allows the GPS library code to read new data from the GPS module and update its state. Since the GPS module is always sending data you have to be careful to constantly read data or else you might start to lose data as buffers are filled.

You can check the `has_fix` property to see if the module has a GPS location fix, and if so there are a host of attributes to read like `latitude` and `longitude` (available in degrees):

```
if not gps.has_fix:
    # Try again if we don't have a fix yet.
    print('Waiting for fix...')
    continue
# We have a fix! (gps.has_fix is true)
# Print out details about the fix like location, date, etc.
print('=' * 40) # Print a separator line.
print('Fix timestamp: {}/{}/{} {:02}:{:02}:{:02}'.format(
    gps.timestamp_utc.tm_mon, # Grab parts of the time from the
    gps.timestamp_utc.tm_mday, # struct_time object that holds
    gps.timestamp_utc.tm_year, # the fix time. Note you might
    gps.timestamp_utc.tm_hour, # not get all data like year, day,
    gps.timestamp_utc.tm_min, # month!
    gps.timestamp_utc.tm_sec))
print('Latitude: {} degrees'.format(gps.latitude))
print('Longitude: {} degrees'.format(gps.longitude))
print('Fix quality: {}'.format(gps.fix_quality))
# Some attributes beyond latitude, longitude and timestamp are optional
# and might not be present. Check if they're None before trying to use!
if gps.satellites is not None:
    print('# satellites: {}'.format(gps.satellites))
if gps.altitude_m is not None:
    print('Altitude: {} meters'.format(gps.altitude_m))
if gps.track_angle_deg is not None:
    print('Speed: {} knots'.format(gps.speed_knots))
if gps.track_angle_deg is not None:
    print('Track angle: {} degrees'.format(gps.track_angle_deg))
if gps.horizontal_dilution is not None:
    print('Horizontal dilution: {}'.format(gps.horizontal_dilution))
if gps.height_geoid is not None:
```

Notice some of the attributes like `altitude_m` are checked to be `None` before reading. This is a smart check to put in your code, because those attributes are sometimes not sent by a GPS module. If an attribute isn't sent by the module it will be given a `None` /null value and attempting to print or read it in Python will fail. The core attributes of

`latitude` , `longitude` , and `timestamp` are usually always available (if you're using the example as-is) but they might not be if you turn off those outputs with a custom NMEA command!

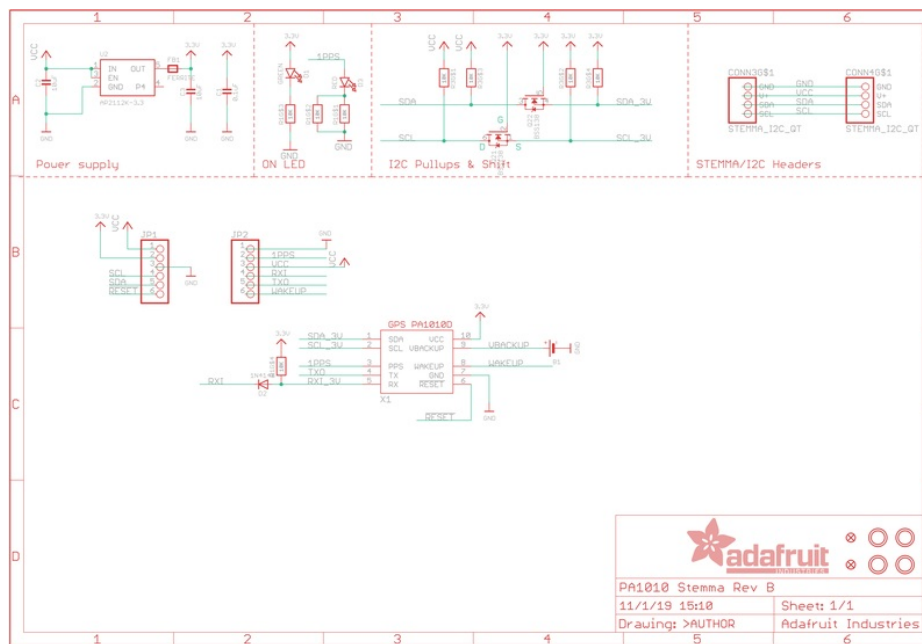
That's all there is to reading GPS location with CircuitPython code!

Downloads

Files:

- PA1010D Datasheet (<https://adafru.it/H3e>)
- EagleCAD PCB files on GitHub (<https://adafru.it/H3f>)
- Fritzing object in the Adafruit Fritzing Library (<https://adafru.it/H3A>)

Schematic



Fab Print

