GPSD Client HOWTO

Eric S. Raymond
<esr@thyrsus.com>
version 1.19, Jul 2015

Table of Contents

Introduction

This document is a guide to interfacing client applications with GPSD. It surveys the available bindings and their use cases. It also explains some sharp edges in the client API which, unfortunately, are fundamental results of the way GPS sensor devices operate, and suggests tactics for avoiding being cut.

Sensor behavior matters

GPSD handles two main kinds of sensors: GPS receivers and AIS receivers. It has rudimentary support for some other kinds of specialized geolocation-related sensors as well, notably compass and yaw/pitch/roll, but those sensors are usually combined with GPS/AIS receivers and behave like them.

In an ideal world, GPS/AIS sensors would be oracles that you could poll at any time to get clean data. But despite the existence of some vendor-specific query and control strings on some devices, a GPS/AIS sensor is not a synchronous device you can query for specified data and count on getting a response back from in a fixed period of time. It gets radio data on its own schedule (usually once per second for a GPS), and emits the reports it feels like reporting asynchronously with variable lag during the following second. **If** it supports query strings, reports from these are intermixed with the regular reports, and usually scheduled at a lower priority (often with a lag of more than a second).

A GPS/AIS receiver, or any sensor that behaves like one, is in effect a datagram emitter similar to a UDP data source; you get no guarantees about sequence or timing at finer resolution than "TPV roughly once per second" (or whatever the main type of report and report interval is).

The only way to simulate synchronous querying of such a sensor is to have an agent between you and it that caches the data coming out of the device; you can then query the agent and expect a reasonably synchronous response from the cache (we support this for GPSes). However, note that this doesn't work for a freshly opened device - there's no cached data.

Consider what this implies for GPses, in particular:

- You can't actually poll them. They report to you over a serial (RS232 or USB-serial) interface at a fixed interval, usually once per second.
- They don't always have a fix to report, because they can only get one when they have satellite lock.
- They may fail to have satellite lock when your skyview is poor. If you are moving through uneven terrain or anywhere with trees or buildings, your skyview can degrade in an instant.
- They may also fail to have lock because they're initially powering on (cold start), or waking from a power-conserving sleep mode (warm start). While most modern GPSes with a good skyview can get satellite lock in 30 seconds from a warm start, a GPS that has been off for a while can take 15 minutes or more to acquire lock

Time to fix after a power-on matters because in many use cases for GPSes they're running off a battery, and you can't afford to keep them powered when you don't actually need location data. This is why GPS sensors are sometimes designed to go to a low-power mode when you close the serial port they're attached to.

AIS receivers have a simpler set of constraints. They report navigational and ID information from any AIS transmitter in line of sight; there are no skyview issues, and they're ready instantly when powered up. Furthermore, they're not normally battery constrained. However, you don't poll them either; they receive information packets over the air and ship them to you at unpredictable intervals over a serial port.

The design of the GPSD reporting protocol surrenders to reality. Its data packets translate the GPS's stream of datagrams into device-type-independent datagrams, but it can't impose timing and sequencing regularities on them that the underlying device doesn't already obey.

What GPSD does, and what it cannot do

GPSD solves some of the problems with GPS/AIS sensors. First: multiplexing; it allows multiple applications to get sensor data without having to contend for a single serial device. Second: coping with the hideous gallimaufry of badly-designed protocols these devices use — regardless of device type, you will get data in a single well-documented format. Third: on operating systems with a hotplug facility (like Linux udev), GPSD will handle all the device management as USB devices are plugged in and unplugged.

What GPSD can't do is pull fix data out of thin air when your device hasn't reported any. Nor is it a magic power supply, so its device management has to be designed around keeping the attached sensors open only when a client application actually needs a fix.

As you'll see, these constraints explain most of the design of the GPSD wire protocol, and of the library APIs your client application will be using.

How the GPSD wire protocol works

While GPSD project ships several library bindings that will hide the details of the wire protocol from you, you'll understand the library APIs better by knowing what a wire-protocol session looks like. After reading this section, you can forget the details about commands and responses and attributes as long as you hold on to the basic logical flow of a session.

Your client library's open function is going to connect a socket to port 2947 on the host your sensors are attached to, usually localhost. On connection, the gpsd daemon will ship a banner that looks something like this:

```
{"class":"VERSION","release":"2.93","rev":"2010-03-30T12:18:17",
                  "proto_major":3,"proto_minor":2}
```

There's nothing mysterious here. Your server daemon is identifying itself with information that may allow a client library to work around bugs or potential incompatibilities produced by upgrades.

To get data from the attached sensors, you need to explicitly tell the daemon you want it. (Remember that it's trying to minimize the amount of time the devices are held open and in a fully powered state.) You do this by issuing a WATCH command:

```
?WATCH={"enable":true,"json":true}
```

This tells the daemon to watch all devices and to issue reports in JSON. It can ship some other protocols as well (notably, NMEA 0183) but JSON is the most capable and usually what you want.

A side effect of the WATCH command is that the daemon will ship you back some information on available devices.

```
{"class":"DEVICES","devices":[{"class":"DEVICE","path":"/dev/ttyUSB0",
                   "activated":1269959537.20,"native":0,"bps":4800,"parity":"N",
                   "stopbits":1,"cycle":1.00}]}
{"class":"WATCH","enable":true,"json":true,"nmea":false,"raw":0,
                 "scaled":false,"timing":false,"pps":false}
```

The DEVICES response tells you what devices are available to the daemon; this list is maintained in a way you as the application designer don't have to care about. The WATCH response will immediately follow and tells you what all your watch request settings are.

Up to this point, nothing has been dependent on the state of the sensors. At this time, it may well be that none of those devices is fully powered up yet. In fact, they won't be, unless another GPSD-enabled application is already watching when you open your connection. If that's the case, you will start seeing data immediately.

For now, though, let's go back to the case where gpsd has to fire up the sensors. After issuing the WATCH response, the daemon opens all of them and watches for incoming packets that it can recognize. **After a variable delay**, it will ship a notification that looks something like this:

```
{"class":"DEVICE","path":"/dev/ttyUSB0","activated":1269960793.97,
                 "driver":"SiRF binary","native":1,"bps":4800,
                 "parity":"N","stopbits":1,"cycle":1.00}
```

This is the daemon telling you that it has recognized a SiRF binary GPS on /dev/ttyUSB0 shipping report packets at 4800 bits per second. This notification is not delayed by the time it takes to achieve satellite lock; the GPS will cheerfully ship packets before that. But it will be delayed by the time required for the daemon to sync up with the GPS.

The GPSD daemon is designed so it doesn't have to know anything about the sensor in advance - not which of a dozen reporting protocols it uses, and not even the baud rate of the serial device. The reason for this agnosticism is so the daemon can adapt properly to anything a hotplug event night throw at it. If you unplug your GPS while your application is running, and then plug one one of a different type, the daemon will cope. Your application won't know the difference unless you have told it to notice device types.

You can even start your application, have it issue a WATCH, realize you forgot to plug in a GPS, and do that. The hotplug event will tell gpsd, which will add the new device to the watched-devices list of every client that has issued a ?WATCH.

In order to make this work, gpsd has a packet sniffer inside it that does autobauding and packet-protocol detection. Normally the packet sniffer will achieve sync in well under a second (my measured times range from 0.10 to 0.53 sec at 4800bps), but it can take longer if your serial traffic is degraded by dodgy cables or electrical noise, or if the GPS is configured to run at an unusual speed/parity/stopbit configuration.

The real point here is that the delay is **variable**. The client library, and your application, can't assume a neat lockstep of request and instant response.

Once you do get your device(s) synced, things become more predictable. The sensor will start shipping fix reports at a constant interval, usually every second, and the daemon will massage them into JSON and pass them up the client to your application.

However, until the sensor achieves satellite lock, those fixes will be "mode 1" - no valid data (mode 2 is a 2D fix, mode 3 is a 3D fix). Here's what that looks like:

```
{"class":"TPV","device":"/dev/ttyUSB0",
               "time":"2010-04-30T11:47:43.28Z","ept":0.005,"mode":1}
```

Occasionally you'll get another kind of sentence, SKY, that reports a satellite skyview. But TPV is the important one. Here's what it looks like when the sensor has a fix to report:

```
{"class":"TPV","time":"2010-04-30T11:48:20.10Z","ept":0.005,
               "lat":46.498204497,"lon":7.568061439,"alt":1327.689,
                "epx":15.319,"epy":17.054,"epv":124.484,"track":10.3797,
                "speed":0.091,"climb":-0.085,"eps":34.11,"mode":3}
```

Note the "mode":3 at the end. This is how you tell that the GPS is reporting a full 3D fix with altitude.

If you have an AIS receiver attached, it too will have been opened and autobauded and protocol-sniffed after your WATCH. The stream of JSON objects will then include things like this:

```
{"class":"AIS","type":5,"repeat":0,"mmsi":351759000,"scaled":true,
               "imo":9134270,"ais_version":0,"callsign":"3FOF8",
               "shipname":"EVER DIADEM",
               "shiptype":"Cargo - all ships of this type",
               "to_bow":225,
               "to_stern":70,"to_port":1,"to_starboard":31,"draught":12.2,
               "epfd":"GPS","eta":"05-15T14:00Z",
               "destination":"NEW YORK","dte":0}
```

When your application shuts down, it can cancel its watch:

```
?WATCH={"enable":false}
```

This will enable the daemon to close devices and conserve power. Supposing you don't do this, the daemon will time out devices with no listeners, so canceling your watch is not strictly necessary. But it is good manners.

Another way to use the daemon is with the ?POLL command To do this, issue

```
?WATCH={"enable":true}
```

This activates all devices without enabling streaming of reports. You can then say "?POLL;" to poll gpsd's recorded data.

```
?POLL;
{"class":"POLL","time":"2012-04-05T15:00:01.501Z","active":1,
    "tpv":[{"class":"TPV","device":"/dev/ttyUSB0","mode":3,"time":"2012-04-
05T15:00:00.000Z","ept":0.005,"lat":40.035083522,"lon":-75.519982905,"alt":166.145,"epx":9.125,"epy":17.778,"
[{"class":"GST","device":"/dev/ttyUSB0","time":"1970-01-
01T00:00:00.000Z","rms":0.000,"major":0.000,"minor":0.000,"orient":0.000,"lat":0.000,"lon":0.000,"alt":0.000

     "sky":[{"class":"SKY","device":"/dev/ttyUSB0","time":"2012-04-
05T15:00:00.000Z","xdop":0.61,"ydop":1.19,"vdop":1.48,"tdop":1.14,"hdop":1.40,"gdop":2.30,"pdop":1.99,"satel
[{"PRN":26,"el":15,"az":49,"ss":29,"used":true},{"PRN":18,"el":62,"az":315,"ss":31,"used":true},
{"PRN":15,"el":60,"az":43,"ss":44,"used":true},{"PRN":21,"el":71,"az":237,"ss":0,"used":false},
{"PRN":27,"el":52,"az":94,"ss":40,"used":true},{"PRN":9,"el":48,"az":136,"ss":33,"used":true},
{"PRN":22,"el":21,"az":291,"ss":36,"used":true},{"PRN":3,"el":8,"az":303,"ss":25,"used":true}]}]}
```

This interface is intended for use with applications like CGI scripts that cannot wait on output from the daemon but must poke it into responding.

If you're a clever sort, you're already wondering what the daemon does if the application at the other end of the client socket doesn't read data out of it as fast as gpsd is shipping it upwards. And the answer is this: eventually the socket buffer fills up, a write from the daemon throws an error, and the daemon shuts down that client socket.

From the point of view of the application, it reads all the buffered data and then gets a read return indicating the socket shutdown. We'll return to this in the discussion of client libraries, but the thing for you to know right now is that this edge case is actually quite difficult to fall afoul of. Total data volume on these sockets is not high. As long as your application checks for and reads socket data no less often than once a second, you won't — and a second is a **lot** of time in which to come back around your main loop.

Interfacing from the client side

The gpsd daemon exports data in three different ways: via a sockets interface, via DBUS broadcasts, and via a shared-memory interface, It is possible one or more of these may be configured out in your installation.

The sockets interface

The GPSD project provides client-side libraries in C, C++, and Python that exercise the sockets export. A Perl module is separately available from CPAN. While the details of these libraries vary, they all have the same two purposes and the same limitations.

One purpose of the libraries is to handle the details of unpacking JSON-based wire-protocol objects into whatever native structure/record feature your application language has. This is particularly important in the C and C++ libraries, because those languages don't have good native support for JSON.

Another purpose is to hide the details of the wire protocol from the application. This gives the GPSD developers room to improve extend the protocol without breaking every client application. Depend on wire-protocol details only at your own risk!

The limitations the libraries have to cope with are the nature of the data flow from the sensors, and the simple fact that they're not necessarily delivering fixes at any given time.

For details of the libraries' APIs, see their reference documentation; the objective of the rest of this section is to teach you the general model of client-side interfacing that they all have to follow because of the way the daemon works.

Each library has the following entry points:

- Open a session socket to the daemon. Named something like "open()".
- Set watch policy. Named something like "stream()"
- Send wire-protocol commands to the daemon. Deprecated; makes your code dependent on the wire protocol. There is no longer a real use case for this entry point; if you think you need no use it, you have probably failed to understand the rest of the interface.
- Blocking check to see if data from the daemon is waiting. Named something like "waiting()" and taking a wait timeout as argument. Note that choosing a wait timeout of less than twice the cycle time of your device will be hazardous, as the receiver will probably not supply input often enough to prevent a spurious error indication. For the typical 1-second cycle time of GPSes this implies a minimum 2-second timeout.
- Blocking read for data from the daemon. Named something like "read()" (this was "poll()" in older versions).
- Close the session socket. Named something like "close()".
- Enable debugging trace messages

The fact that the data-waiting check and the read both block means that, if your application has to deal with other input sources than the GPS, you will probably have to isolate the read loop in a thread with a mutex lock on the gps_data structure.

Here is a complete table of the binding entry points:

Table 1. Entry points in client bindings

C	C++	Python	Function
gps_open()	gpsmm.gpsmm()	gps.__init__()	In OO languages the client class initializer opens the daemon socket.
gps_send()	gpsmm.send()	gps.send()	Send wire-protocol commands to the daemon. Deprecated and unstable.
gps_stream()	gpsmm.stream()	gps.stream()	Set watch policy. What you should use instead of send().
gps_waiting()	gpsmm.waiting()	gps.waiting()	Blocking check with timeout to see if input is waiting.
gps_read()	gpsmm.read()	gps.read()	Blocking read for data from the daemon.
gps_unpack()		gps.unpack()	Parse JSON from a specified buffer into a session structure
gps_close()	gpsmm.~gpsmm()	gps.close()	Close the daemon socket and end the session.
gps_data()	gpsmm.data()	gps.data()	Get the contents of the client buffer.
gps_enable_debug()	gpsmm_enable_debug()		Enable debug tracing. Only useful for GPSD developers.
gps_clear_fix()	gpsmm.clear_fix()		Clear the contents of the fix structure.

The tricky part is interpreting what you get from the blocking read. The reason it's tricky is that you're not guaranteed that every read will pick up exactly one complete JSON object from the daemon. It may grab one response object, or more than one, or part of one, or one or more followed by a fragment.

What the library does on each read is this: get what it can from the socket, append that to a private buffer, and then consume as many JSON objects from the front of the buffer as it can. Any incomplete JSON is left in the private buffer to be completed and unpacked on a later go-round.

In C, the library "consumes" a JSON object by unpacking its content into a blackboard structure passed to the read entry point by address. The structure contains a state-flag mask that you can (and should!) check so you'll know which parts of the structure contain valid data. It is safe to do nothing unless the PACKET_SET mask bit is on, which is the library's way of telling you that at least one complete JSON response has arrived since the last read.

Data may accumulate on the blackboard over multiple reads, with new TPV reports overwriting old ones; it is guaranteed that overwrites are not partial. Expect this pattern to be replicated in any compiled language with only fixed-extent structures.

In Python and Perl the read entry point returns an object containing accumulated data. The state-flag mask is still useful for telling you which parts contain data, and there is still a PACKET_SET bit. Expect this pattern to be replicated in other dynamic OO languages when we support them.

The C++ binding is a very thin wrapper around the C. You get back an object, but it's just a reference to the C blackboard structure. There's no unpack() method because it doesn't fit the gpsmm object's RAII model.

All bindings will throw a recognizable error from the read entry point when the socket is closed from the daemon side.

> **Warning** The timing of your read loop is important. When it has satellite lock, the daemon will be writing into its end of the socket once per whatever the normal reporting-cycle time of your device is - for a GPS normally one peer second. **You must poll the socket more often that that.**

If necessary, spawn a worker thread to do this, mutex-locking the structure where it outs the reports. If you don't do this, data will back up in your socket buffers and position reports will be more and more delayed until the socket FIFO fills, at which point the daemon will conclude the client has died and drop the connection.

AIVDM clients have a longer maximum allowable poll interval, but a problem of a different kind. you have the problem that later sentences of (say) Type 1 don't obsolete the data in earlier ones. This is a problem, because the library is designed so that read calls pull any JSON reports waiting from the daemon and interpret them all.

To avoid losing data, you want to poll the daemon more often than once per two seconds (that being the minimum transmission period for the most frequently shipped sentences, Type 1/2/3). That way the read buffer will never contain both a message and a later message of the same type that steps on it.

Shared-memory interface

Whenever gpsd recognizes a packet from any attached device, it writes the accumulated state from that device to a shared memory segment. The C and C++ client libraries shipped with GPSD can read this segment.

The API for reading the segment uses the same gps_open(), gps_read() and gps_close() entry points as the sockets interface. To enable using shared memory instead, it is only necessary to use the macro constant GPSD_SHARED_MEMORY as the host argument of gps_open().

The gps_stream(), gps_send(), gps_waiting(), and gps_data() entry points are not available with this interface. You cannot set a device filter on it. You will not get device activation or deactivation notices through it. And, of course, it is only good for local and not networked access. Its main advantage is that it is very fast and lightweight, especially suitable for use in low-power embedded deployments with a single device on a fixed port and the sockets interface configured out.

Under the shared-memory interface, gps_read() after a successful gps_open() will always return with data; its return is the size of a struct gps_data_t in bytes. The *gps_fd* member of the struct gpsdata instance handed to you will always be -1. The PACKET_SET flag will always be asserted. The other flag bits in the *set* member will tell you what data is updated in the instance, just as in the sockets interface.

The shared-memory interface is not yet available from Python.

D-Bus broadcasts

If your system supports D-Bus, gpsd broadcasts a signal with path /org/gpsd, interface "org.gpsd", and name "fix" whenever it received a position report from any device attached to it. See the gpsd(8) manual page for details of the binary payload layout.

C Examples

The source distribution includes two example clients in C; gpxlogger.c and cgps.c.

gpxlogger.c illustrates the simplest possible program flow; open, followed by stream, followed by the library main loop.

cgps.c shows what an interactive application using the library and also hw processing user commands works. Note the use of the curses nodelay function to ensure that wgetch() does not block the GPS polling loop.

C++ examples

The following code skeleton implements a C++ client:

```
int main(void)
{
    gpsmm gps_rec("localhost", DEFAULT_GPSD_PORT);

    if (gps_rec.stream(WATCH_ENABLE|WATCH_JSON) == NULL) {
```

```
        cerr << "No GPSD running.\n";
        return 1;
    }

    for (;;) {
        struct gps_data_t* newdata;

        if (!gps_rec.waiting(50000000))
          continue;

        if ((newdata = gps_rec.read()) == NULL) {
            cerr << "Read error.\n";
            return 1;
        } else {
            PROCESS(newdata);
        }
    }
    return 0;
}
```

Note the absence of explicit open and close methods. The object interface is designed on the RAII (Resource Acquisition Is Initialization) model; you close it by deallocating it.

Look at test_libgpsmm.cpp in the distribution for a full example.

Python examples

There's a very simple Python example analogous to gpxlogger attached to the source code for the gps.py library.

The heart of it is this code:

```
    session = gps(**opts)
    session.stream(WATCH_ENABLE|WATCH_NEWSTYLE)
    for report in session:
        print report
```

If you need to intersperse other processing in a main event loop, like this:

```
session = gps(mode=WATCH_ENABLE)
try:
    while True:
        # Do stuff
        report = session.next()
        # Check report class for 'DEVICE' messages from gpsd.  If
        # we're expecting messages from multiple devices we should
        # inspect the message to determine which device
        # has just become available.  But if we're just listening
        # to a single device, this may do.
        if report['class'] == 'DEVICE':
            # Clean up our current connection.
            session.close()
            # Tell gpsd we're ready to receive messages.
            session = gps(mode=WATCH_ENABLE)
        # Do more stuff
except StopIteration:
    print "GPSD has terminated"
```

Each call to the iterator yields a report structure until the daemon terminates, at which point the iterator next() method will raise StopIteration and the loop will terminate.

The report object returned by next() can be accessed either as a dictionary or as an object. As a dictionary, it is the raw contents of the last JSON response re-encoded in plain ASCII. For convenience, you may also access it as an object with members for each attribute in the dictionary. It is especially useful to know that the object will always have a "class" member giving the response type (TPV, SKY, DEVICE, etc.) as a string.

For more interesting examples integrated with X and GTK, see xgps and xgpsspeed.

Other Client Bindings

There are a couple of client bindings for GPSD that are maintained separately from the GPSD distribution. We don't try to document their APIs here, but just provide pointers to them.

Java

There is a Java binding, described at http://gpsd4java.forge.hoegergroup.de/ This binding is available at maven central. See that web page for how to use it in a maven build.

Perl

There's a Perl client library at http://search.cpan.org/dist/Net-GPSD3/

Backward Incompatibility and Future Changes

The C/C++ binding makes available two preprocessor symbols, GPSD_API_MAJOR_VERSION and GPSD_API_MINOR_VERSION, in gps.h. The Python module has corresponding symbols.

In major versions before 5:

- gps_open() didn't take a third argument; instead, it returned malloc storage.
- The *read()* method in various bindings was named *poll()*, blocked waiting for input, and had a different return convention. The name *poll()* will at some point be reintroduced as an interface to the wire-protocol POLL command.
- Clients needed to define a hook for client-side logging if they didn't want code in netlib.c and libgps_core.c to occasionally send messages to stderr. This requirement is now gone.
- There was a set_raw_hook() method in the C and Python bindings, now gone. C clients should call gps_data(); the buffer is available directly in Python.

Version 1.19
Last updated 2016-08-31 18:26:00 EDT