GE863-PRO3 GPS Package User Guide

1VV0300857 Rev.0 – 2009-10-13

Making machines talk.

Disclaimer

Applicable Products

PRODUCT
GE863-PRO[3]

Linux SW Version
04.0004 or higher

Contents

1. Introduction

1.1. Scope

This user guide serves the following purposes:

- Describe the GPSD Package for GPS receivers' control under Linux.
- Describe how software developers can use GPSD APIs to create GPS client applications for GE863-PRO^3 for controlling GPS receivers.

This document refers to GPSD 2.37 version.

1.2. Audience

This User Guide is intended for customers who want to develop GPS client applications for GE863-PRO^3.

1.3. Contact Information, Support

For general contact, technical support, to report documentation errors and to order manuals, contact Telit's Technical Support Center (TTSC) at:

TS-EMEA@telit.com
TS-NORTHAMERICA@telit.com
TS-LATINAMERICA@telit.com
TS-APAC@telit.com

Alternatively, use:
http://www.telit.com/en/products/technical-support-center/contact.php
For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:
http://www.telit.com
To register for product news and announcements or for product questions contact Telit Technical Support Center (TTSC).
Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.
Telit appreciates feedback from the users of our information.

1.4. Open Source Licenses

1.4.1. GPSD License

1.5. Product Overview

The GE863-PRO^3 module contains a fully featured GSM/GPRS communications section, compatible with the other Telit GSM/GPRS modules, but also incorporates a standalone ARM9 CPU and memories, dedicated to user applications.

This eliminates the need for an external host CPU in many applications, bringing true real-time and multi tasking capabilities to an embedded module.

1.6. Document Organization

This manual contains the following chapters:

- "Chapter 1, Introduction" provides a scope for this manual, target audience, technical contact information, and text conventions.
- "Chapter 2, Overview" provides a brief description of how GPS receivers relay GPS data.
- "Chapter 3, GPSD Package" describes GPSD Package detailing each of its components.
- "Chapter 4, Appendix A" provides a description of request/answer protocol used by gpsd clients.
- "Chapter 5, Appendix B" provides GPS Acronyms, Abbreviations and Glossary.

How to Use

If you are new to this product, it is recommended to start by reading this document and the following, in order to understand the concepts and specific features provided by the built in software of the GE863-PRO^3:

- TelitGE863PRO3 EVK User Guide 1VV0300776
- GE863PRO3 Linux Development Environment User Guide 1VV0300780
- GE863PRO3 Linux SW UserGuide 1vv0300781

1.7. Text Conventions

This section lists the paragraph and font styles used for the various types of information presented in this user guide.

Format	Content
`Courier New`	Linux shell commands, filesystem paths and C source code examples

All dates are in ISO 8601 format, i.e. YYYY-MM-DD.

1.8. Related Documents

The following documents are related to this user guide:

[1] TelitGE863PRO3 Linux SW User Guide 1vv0300781
[2] TelitGE863PRO3 Hardware User Guide 1vv0300773a
[3] TelitGE863PRO3 EVK User Guide 1VV0300776
[4] TelitGE863PRO3 Linux Development Environment 1VV0300780

All documentation can be downloaded from Telit's official web site www.telit.com if not otherwise indicated.

1.9. Document History

Revision	Date	Changes
ISSUE #0	2009-10-13	First issue

2. Overview

In order to relay computed GPS variables such as position, velocity, course etc. to a peripheral (e.g. computer, screen, transceiver), GPS modules use a serial interface. The most important elements of receiver information are broadcasted via this interface in a special data format. This format is standardized by the National Marine Electronics Association (NMEA) to ensure that data exchange takes place without any problems. Nowadays, data is relayed according to the NMEA-0183 specification.
NMEA data stream must be parsed and interpreted so that all the relayed information can be retrieved and therefore used. The open source GPSD Package offers all the functionalities to perform management and control of the GPS receiver and NMEA data decoding under Linux OS.

3. GPSD Package

GPSD is an Open Source Project (http://gpsd.berlios.de/#documentation) providing the following features:

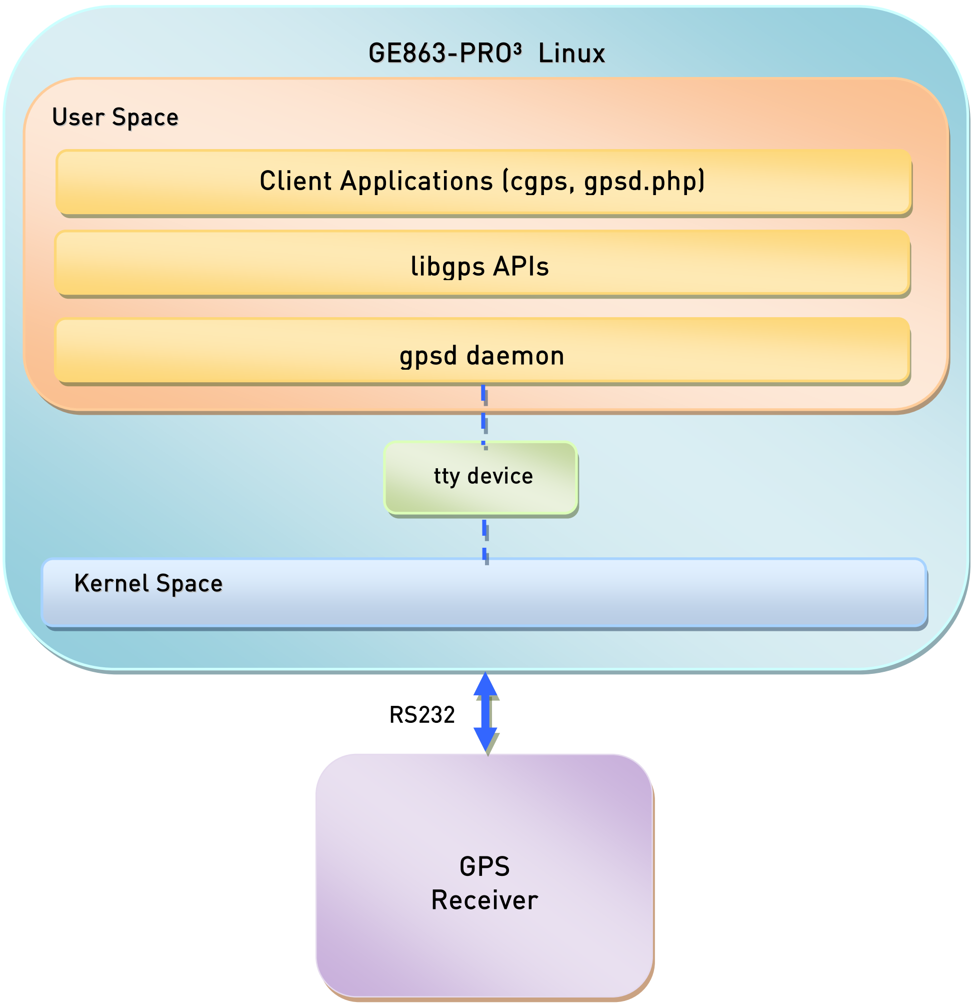
- Management and control of the GPS receiver attached to the controlling system by means of a RS-232 serial port
- Retrieving, parsing and decoding of NMEA protocol and retrieving of the following GNSS information:

  - Fix Status (if the GPS receiver has acquired a 2D/3D fix or not)
  - Latitude
  - Longitude
  - Altitude
  - UTC Time
  - Speed
  - Heading
  - Climb Speed
  - Estimated position errors based on HDOP, VDOP and PDOP
  - Visible Satellites and for each of them:
    - Elevation
    - Azimuth
    - SNR

GE863-Pro^3 GPSD Package is made up of the followings:

- **gpsd daemon,** for management and control of the GPS receiver;
- **libgps library APIs**, to ease development of GPS client applications for GE863-PRO^3 interfacing to the gpsd control daemon;
- **cgps**, test client application to show, using libgps APIs, the GPS information received form a GPS module (e.g. Fix Status, Latitude, Longitude, Altitude, UTC Time, Speed, Heading and Climb Speed).
- **gpsd.php**, php test client to show, interfacing to the gpsd control daemon, the calculated point onto a web page with Google Maps using Google APIs

The figure below shows the software framework for GPSD Package.



3.1. Package Installation

If you don't have GPSD Package for GE863-PRO³ yet, you can download it from Telit's official web site Download Zone http://www.telit.com/en/products/download-zone.php.

gpsd_pro3.tar and libgps_pro3.tar files must be downloaded to install all GPSD Package components.

3.1.1. GE863-PRO^3

Once gpsd_pro3.tar file has been downloaded, it can be copied into the GE863-PRO^3 root folder "/" as described in [4].

To install gpsd_pro3.tar, from the root folder "/" type:

```
# tar –zxvf gpsd_pro3.tar
```

This will install the following GPSD Package components:

- gpsd daemon into /sbin
- cgps client application into /bin
- gpsd.php and gpsd_config.inc into /var/www

Now remove gpsd_pro3.tar file from target's filesystem:

```
# rm gpsd_pro3.tar
```

3.1.2. Development Environment

Once libgps_pro3.tar file has been downloaded, it can be copied into the coLinux root folder "/" (refer to [4] for further information about Telit GE863-PRO^3 Development Environment).

To install libgps_pro3.tar, from the root folder "/" type:

```
# tar –zxvf libgps_pro3.tar
```

This will install the following GPSD Package components:

- libgps.a into /opt/crosstools/telit/lib
- gps.h into /opt/crosstools/telit/include

Now remove libgps_pro3.tar file from coLinux's filesystem:

```
# rm libgps_pro3.tar
```

3.2. gpsd daemon

The gpsd daemon is a Linux service that monitors one GPS receiver attached to the GE863-Pro^3 through serial port, making all data on the location/course/velocity of the sensor available to be queried on TCP port 2947 (default port) of the GE863-Pro^3. With this control daemon, multiple GPS client applications, such as navigational software, can share access to the GPS receiver without contention or loss of data.

Please note that loopback interface must be enabled to be able to use gpsd and client applications, therefore type:

```
# ifconfig lo up
```

Assuming a GPS receiver is connected to GE863-Pro^3 through RS232 port corresponding to /dev/ttyS1 device (see [3]) gpsd can be run simply typing:

```
# gpsd –n –N –D2 /dev/ttyS1
```

All gpsd command line options are shown below.

Synopsis:

gpsd [-b] [-n] [-N] [-D <n>] [-F <sockfile>] [-P <pidfile>] [-S <port>] [-h] <device>

Parameters:

-b Broken-device-safety, otherwise known as read-only mode. Some popular bluetooth and USB receivers lock up or become totally inaccessible when probed or reconfigured. This switch prevents gpsd from writing to a receiver. This means that gpsd cannot configure the receiver for optimal performance, but it also means that gpsd cannot break the receiver. A better solution would be for bluetooth to not be so fragile. A platform independent method to identify serial-over-bluetooth devices would also be nice.

-n Don't wait for a client to connect before polling whatever GPS is associated with it. It is thought that some GPSes go to a standby mode (drawing less power) before the host machine asserts DTR, so waiting for the first actual request might save battery power on portable equipment. This option combines well with -D2 to enable monitoring of the GPS data stream.

-N Don't daemonize; run in foreground. Also suppresses privilege-dropping. This switch is mainly useful for debugging. Its meaning may change in future versions.

-F <sockfile> Create a control socket for device addition and removal commands. You must specify a valid pathname on your local filesystem; this will be created as a Unix-domain socket to which you can write commands that edit the daemon's internal device list.

-P <pidfile> Specify the name and path to record the daemon's process ID.

-D <n> Set debug level (default 0). gpsd reports, at debug levels 2 and above, incoming sentence and actions to standard error if gpsd is in the foreground (-N) or to syslog if in the background.

-S <port> Set TCP/IP port on which to listen for GPSD clients (default is 2947).

-h Display help message and terminate.

-V Dump version and exit.

<device> Normally, a data source is the name of a local serial device (e.g. /dev/ttyS1) or a list of devices (e.g. /dev/ttyS1, /dev/ttyS2) from which the daemon may expect GPS data.

A data source name may also be a URL pointing to a specific differential-GPS service (DGPSIP server or Ntrip broadcaster). If the URL starts with "ntrip://" Ntrip will be used; if the URL starts with "dgpsip://", DGPSIP will be used. Gpsd also defaults to DGPSIP if no protocol is defined. For Ntrip services that require authentication, a prefix of the form "username:password@" can be added before the name of the Ntrip broadcaster. If a suffix of the service name begins with ":" it is interpreted as a port number, overriding the default IANA-assigned port of 2101. For Ntrip service you also need to specify which stream to use; the stream is given in the form "streamname". So, an example DGPSIP URL could be "dgpsip://dgpsip.example.com" and a Ntrip URL could be:

"ntrip://foo:bar@ntrip.example.com:80/example-stream"

Internally, the daemon maintains a device list holding the pathnames of GPSes known to the daemon. Initially, this list is the list of device-name arguments specified on the command line. That list may be empty, in which case the daemon will have no devices on its search list until they are added by a control-socket command. Daemon startup will abort with an error if neither any devices nor a control socket are specified.

Once gpsd has successfully started, client applications can connect to it through the chosen TCP port, using libgps APIs as described in paragraph 3.3 to show all computed GNSS information.
gpsd clients use a request/answer protocol to communicate with gpsd and retrieve GNSS information (longitude, latitude, altitude, etc..): refer to Appendix A for a complete description of gpsd request/answer protocol.

3.3. libgps library APIs

libgps is a service library which interfaces with gpsd to allow client applications to retrieve GNSS data.
libgps uses a gps_data_t data structure to store all GNSS information and whose fields are updated upon queries performed by the client application.
To develop an application using libgps, use the linker option –lgps and include gps.h header file. libpthread and libm have to be linked also. Please see [4] for further information on how to create a C source project for GE863-Pro^3.

3.3.1. Structures

3.3.1.1. gps_data_t

gps_data_t is the GPS-data structure which holds all the data collected by the GPS and whose fields are updated through the functions described later on.

```
struct gps_data_t {
    gps_mask_t set;
          /* has field been set since this was last cleared? */
#define ONLINE_SET      0x00000001u
#define TIME_SET 0x00000002u
#define TIMERR_SET      0x00000004u
#define LATLON_SET      0x00000008u
#define ALTITUDE_SET    0x00000010u
#define SPEED_SET       0x00000020u
#define TRACK_SET       0x00000040u
#define CLIMB_SET       0x00000080u
#define STATUS_SET      0x00000100u
#define MODE_SET 0x00000200u
#define HDOP_SET        0x00000400u
#define VDOP_SET        0x00000800u
#define PDOP_SET        0x00001000u
#define TDOP_SET 0x00002000u
#define GDOP_SET 0x00004000u
#define DOP_SET  (HDOP_SET|VDOP_SET|PDOP_SET|TDOP_SET|GDOP_SET)
#define HERR_SET 0x00008000u
#define VERR_SET 0x00010000u
#define PERR_SET 0x00020000u
#define ERR_SET  (HERR_SET | VERR_SET | PERR_SET)
#define SATELLITE_SET   0x00040000u
#define PSEUDORANGE_SET 0x00080000u
#define USED_SET 0x00100000u
#define SPEEDERR_SET    0x00200000u
#define TRACKERR_SET    0x00400000u
#define CLIMBERR_SET    0x00800000u
#define DEVICE_SET      0x01000000u
#define DEVICELIST_SET  0x02000000u
#define DEVICEID_SET    0x04000000u
#define ERROR_SET       0x08000000u
```

```
#define CYCLE_START_SET 0x10000000u
#define RTCM_SET 0x20000000u
#define FIX_SET
  (TIME_SET|MODE_SET|TIMERR_SET|LATLON_SET|HERR_SET|ALTITUDE_SET|VERR_
SET|TRACK_SET|TRACKERR_SET|SPEED_SET|SPEEDERR_SET|CLIMB_SET|CLIMBERR_SE
T)
    double online;
                       /* NZ if GPS is on line, 0 if not.
                         *
                         * Note: gpsd clears this flag when sentences
                         * fail to show up within the GPS's normal
                         * send cycle time. If the host-to-GPS
                         * link is lossy enough to drop entire
                         * sentences, this flag will be
                         * prone to false negatives.
                         */

    struct gps_fix_t    fix;          /* accumulated PVT data */

    double separation;                /* Geoidal separation, MSL -
WGS84 (Meters) */

    /* GPS status -- always valid */
    int    status;             /* Do we have a fix? */
#define STATUS_NO_FIX   0      /* no */
#define STATUS_FIX             1      /* yes, without DGPS */
#define STATUS_DGPS_FIX 2      /* yes, with DGPS */

    /* precision of fix -- valid if satellites_used > 0 */
    int satellites_used /* Number of satellites used in solution */
    int used[MAXCHANNELS];

  /* Store the status for each visible satellite, i.e. if it
         used for the solution or not (1,0 values respectively);
  therefore, since visible satellites PRNs are stored into
         PRN[MAXCHANNELS], by accessing to
         used[MAXCHANNELS] with the same index
         (i=0...MAXCHANNELS-1) we are able to know
                       which satellite is used for GPS solution.*/
    double pdop, hdop, vdop, tdop, gdop;     /* Dilution of precision
*/

    /* redundant with the estimate elments in the fix structure */
    double epe;/* spherical position error, 95% confidence (meters)  */

    /* satellite status -- valid when satellites > 0 */
    int satellites;                   /* # of satellites in view */
    int PRN[MAXCHANNELS];             /* PRNs of satellite */
    int elevation[MAXCHANNELS];       /* elevation of satellite */
    int azimuth[MAXCHANNELS];         /* azimuth */
    int ss[MAXCHANNELS];              /* signal-to-noise ratio (dB) */

    /* compass status -- TrueNorth (and any similar) devices only */
    char headingStatus;
    char pitchStatus;
    char rollStatus;
    double horzField;   /* Magnitude of horizontal magnetic field */

    /* where and what gpsd thinks the device is */
    char gps_device[PATH_MAX];        /* only valid if non-null. */
    char *gps_id;                     /* only valid if non-null. */
    unsigned int baudrate, parity, stopbits;/* RS232 link parameters */
    unsigned int driver_mode;
                        /* whether driver is in native mode or not */

    /* RTCM-104 data */
    struct rtcm_t       rtcm;

    /* device list */
    int ndevices;                     /* count of available devices */
    char **devicelist;                /* list of pathnames */

    /* profiling data for last sentence */
    bool profiling;                           /* profiling enabled? */
    char tag[MAXTAGLEN+1];     /* tag of last sentence processed */
    size_t sentence_length;    /* character count of last sentence */
    double sentence_time;                     /* sentence timestamp */
    double d_xmit_time;        /* beginning of sentence transmission */
    double d_recv_time;        /* daemon receipt time (-> E1+T1) */
    double d_decode_time;      /* daemon end-of-decode time (-> D1) */
    double poll_time;                 /* daemon poll time (-> W) */
    double emit_time;                 /* emission time (-> E2) */
    double c_recv_time;               /* client receipt time (-> T2) */
    double c_decode_time;      /* client end-of-decode time (-> D2) */
    double cycle, mincycle;    /* refresh cycle time in seconds */

    /* these members are private */
    int gps_fd;                /* socket or file descriptor to GPS */
    void (*raw_hook)(struct gps_data_t *, char *, size_t len, int
level);  /* Raw-mode hook for GPS data. */
    void (*thread_hook)(struct gps_data_t *, char *, size_t len, int
level); /* Thread-callback hook for GPS data. */
};
```

struct rtcm_t GPS-data structure is used to store all the RTCM-104 data, when the GPS receiver use RTCM-104 source for differential corrections. Please note that not all GPS receivers use RTCM-104 differential corrections.

```
struct rtcm_t {
    /* header contents */
    unsigned type;       /* RTCM message type */
    unsigned length;     /* length (words) */
    double   zcount;     /* time within hour: GPS time, no leap secs */
    unsigned refstaid;   /* reference station ID */
    unsigned seqnum;     /* nessage sequence number (modulo 8) */
    unsigned stathlth;   /* station health */

    /* message data in decoded form */
    union {
  struct {
      unsigned int nentries;
      struct rangesat_t {               /* data from messages 1 & 9 */
         unsigned ident;                       /* satellite ID */
         unsigned udre;                 /* user diff. range error */
         unsigned issuedata;            /* issue of data */
         double rangerr;                       /* range error */
         double rangerate;              /* range error rate */
      } sat[MAXCORRECTIONS];
  } ranges;
  struct {                              /* data for type 3 messages */
      bool valid;                       /* is message well-formed? */
      double x, y, z;
  } ecef;
  struct {                              /* data from type 4 messages */
      bool valid;                       /* is message well-formed? */
      enum {gps, glonass, unknown} system;
      enum {local, global, invalid} sense;
      char datum[6];
      double dx, dy, dz;
  } reference;
  struct {                              /* data from type 5 messages */
      unsigned int nentries;
      struct consat_t {
         unsigned ident;                       /* satellite ID */
         bool iodl;                     /* issue of data */
         unsigned int health;           /* is satellite healthy? */
#define HEALTH_NORMAL   (0)    /* Radiobeacon operation normal */
#define HEALTH_UNMONITORED     (1) /* No integrity monitor operating */
#define HEALTH_NOINFO          (2)    /* No information available */
#define HEALTH_DONOTUSE        (3)    /* Do not use this radiobeacon */
         int snr;                       /* signal-to-noise ratio, dB */
#define SNR_BAD  -1                     /* not reported */
         unsigned int health_en;               /* health enabled */
```

```
         bool new_data;              /* new data? */
         bool los_warning;           /* line-of-sight warning */
         unsigned int tou;           /* time to unhealth, seconds */
      } sat[MAXHEALTH];
  } conhealth;
  struct {                                  /* data from type 7
messages */
      unsigned int nentries;
      struct station_t {
         double latitude, longitude;        /* location */
         unsigned int range;         /* range in km */
         double frequency;           /* broadcast freq */
         unsigned int health;        /* station health */
         unsigned int station_id;           /* of the transmitter */
         unsigned int bitrate;       /* of station transmissions */
      } station[MAXSTATIONS];
  } almanac;
  /* data from type 16 messages */
  char message[(RTCM_WORDS_MAX-2) * sizeof(isgps30bits_t)];
  /* data from messages of unknown type */
  isgps30bits_t words[RTCM_WORDS_MAX-2];
    } msg_data;
};
```

3.3.2. Functions

3.3.2.1. gps_open()

gps_open() initializes a GPS-data structure to hold the data collected by the GPS, and returns a socket attached to **gpsd**.

Prototype
struct gps_data_t ***gps_open**(const char *host, const char *port)

Arguments
host – host address to connect to
port – host port to be used for TCP connection

Return value
A pointer to a struct gps_data_t on success
NULL on errors. errno is set depending on the error returned from the the socket layer; see *gps.h* for values and explanations

Example
```
struct gps_data_t *gpsdata;    /* Struct for gps data */
char *server = NULL;           /* *server=NULL => 127.0.0.1 */
char *port = "2947";
char *err_str = NULL;
```

```
gpsdata = gps_open(server, port);

if (!gpsdata)
{
  switch ( errno ) {
  case NL_NOSERVICE:  err_str = "can't get service entry"; break;
  case NL_NOHOST:     err_str = "can't get host entry"; break;
  case NL_NOPROTO:    err_str = "can't get protocol entry"; break;
  case NL_NOSOCK:     err_str = "can't create socket"; break;
  case  NL_NOSOCKOPT:    err_str  =  "error  SETSOCKOPT  SO_REUSEADDR";
break;
  case NL_NOCONNECT:  err_str = "can't connect to host"; break;
  default:                  err_str = "Unknown"; break;
  }

  (void)fprintf(  stderr,  "gps_client:  no  gpsd  running  or  network
error: %d, %s\n", errno, err_str);
  exit(2);
}
```

3.3.2.2. gps_set_raw_hook()

gps_set_raw_hook() takes a function you specify and run it (synchronously) on the raw data pulled by a **gps_query()** or **gps_poll()** call. The arguments passed to this hook will be a pointer to a structure containing parsed data, and a buffer containing the raw gpsd response.

Prototype
void **gps_set_raw_hook**(struct gps_data_t *gpsdata, void (*hook)(struct gps_data_t *sentence, char *buf))

Arguments
gpsdata – pointer to the GPS data structure used to store GPS information
hook - function to be run synchronously

Return value
None

Example
```
struct gps_data_t *gpsdata;    /* Struct for gps data */
void update_gps_panel(struct gps_data_t *gpsdata, char *message);

int main()
{
  …..
  gps_open(server, port);
  …..
  gps_set_raw_hook(gpsdata, update_gps_panel);
```

```
  (void)gps_query(gpsdata, "q\n");
  …..
  gps_close(gpsdata);
  …..
}
```

3.3.2.3. gps_set_callback()

gps_set_callback() takes a function you specify and runs it asynchronously each time new data arrives from gpsd, using POSIX threads. Actually gps_set_callback() creates a thread through the pthread_create() function and stores thread's ID. For example, you can call gps_set_callback(gpsdata, my_function, handler) once in your program, and from there on your gps data structure will be parsed by your my_function() each time new data are available. my_function() could change some global variables in your program based on received data; it is your responsibility to ensure that your program uses mutexes or other mechanisms to avoid race conditions.

Prototype
int **gps_set_callback**(struct gps_data_t *gpsdata, void (*callback)(struct gps_data_t *sentence, char *buf), pthread_t *handler)

Arguments
gpsdata – pointer to the GPS data structure used to store GPS information
callback – function to be run asynchronously
handler – thread handler

Return value
Same as the return value of pthread_create() function:
If successful, the *pthread_create*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

Example
```
struct gps_data_t *gpsdata;    /* Struct for gps data */
void update_gps_panel(struct gps_data_t *gpsdata, char *message);

int main()
{
  pthread_t threadID;
  …..
  gps_open(server, port);
  …..
  If ( gps_set_callback(gpsdata, update_gps_panel, &threadID) != 0 )
  {
         /* Error Management */
  }

  (void)gps_poll(gpsdata);
  …..
  gps_close(gpsdata);
  …..

}
```

3.3.2.4. gps_query()

gps_query() writes a command to the daemon, accepts a one-line response, and updates parts of the GPS-data structure that correspond to data changed since the last call. The one-line response from gpsd is written into the char *buf of the callback registered with gps_set_raw_hook() or gps_set_callback(). The user can therefore either parse the one-line response or access to the updated fields of the GPS-data structure depending on its own needs (please have a look to the examples below).

Prototype
int **gps_query**(struct gps_data_t *gpsdata, const char *fmt, ...)

Arguments
gpsdata – pointer to the GPS data structure used to store GPS information
fmt – must be a format string containing letters which are exactly the ones from the command set accepted by gpsd daemon described in Appendix A. It may have % elements as for sprintf, which will be filled in from any following arguments.

Return value
0 if successful, -1 otherwise

Example 1
The following example registers the update_probe() callback through the gps_set_raw_hook() API and sends the "i" command to identify GPS through the gps_query() API. The update_probe() callback is called when gpsd answers: this callback function parses the char *message buffer containing the gpsd one-line response.

```
char gps_type[26];
struct gps_data_t *gpsdata;                  // Struct for gps data

void update_probe(struct gps_data_t *gpsdata, char *message);
// Callback function to be
                // registered with

gps_set_raw_hook()

int main()
{
  char cmd1 = 'i';     // identify the GPS – see Appendix A
  ….
  gps_open(server, port);
  …..
  /* Register the callback function */
  gps_set_raw_hook(gpsdata, (void*)update_probe);

  /* Send the “i” command to gpsd to identify GPS */
  if(gps_query(gpsdata, "%c\n", cmd1) != 0)
  {
         /* Error Management */
  }
  ….
  gps_close(gpsdata);
  …..
}

void update_probe(struct gps_data_t *gpsdata, char *message)
{
  assert(message != NULL);
  memset(gps_type, '\0', sizeof(gps_type));

  if(strncmp(message,"GPSD,I=",6) == 0)
  {
         message+=7;
         (void)strlcpy(gps_type, message, sizeof(gps_type));
  }
}
```

Example 2

The following example registers the update_probe() callback through the gps_set_raw_hook() API and sends the “p” command to retrieve position (latitude, longitude) through the gps_query() API. The update_probe() callback is called when gpsd answers: this callback function perform a direct access to the updated fields of the GPS-data structure.

```
char gps_type[26];
struct gps_data_t *gpsdata;                  // Struct for gps data

void update_probe(struct gps_data_t *gpsdata, char *message);
// Callback function to be
                // registered with

gps_set_raw_hook()
int main()
{
  char cmd1 = ‘p’;      // retrieve position – see Appendix A
  ….
  gps_open(server, port);
  …..
  /* Register the callback function */
  gps_set_raw_hook(gpsdata, (void*)update_probe);
  /* Send the “p” command to gpsd retrieve position */
  if(gps_query(gpsdata, "%c\n", cmd1) != 0)
  {
         /* Error Management */
  }
  ….
  gps_close(gpsdata);
  …..
}

void update_probe(struct gps_data_t *gpsdata, char *message)
{
  char scr[128];

  /* Fill in the latitude. */
  if (isnan(gpsdata->fix.latitude) == 0)
  {
         (void)snprintf(scr, sizeof(scr), "%s %c",
                deg_to_str(deg_type,fabs(gpsdata->fix.latitude)),
                       (gpsdata->fix.latitude < 0) ? 'S' : 'N');
  } else
         (void)snprintf(scr, sizeof(scr), "n/a");
  printf("Latitude:\t%s\n", scr);

  /* Fill in the longitude. */
  if (isnan(gpsdata->fix.longitude) == 0)
  {
         (void)snprintf(scr, sizeof(scr), "%s %c",
                deg_to_str(deg_type,  fabs(gpsdata->fix.longitude)),
                       (gpsdata->fix.longitude < 0) ? 'W' : 'E');
  } else
         (void)snprintf(scr, sizeof(scr), "n/a");
  printf("Longitude:\t%s\n", scr);
}
```

3.3.2.5. gps_poll()

gps_poll() accepts a response, or sequence of responses, from the daemon and interprets it as though it were a query response.

Prototype
int **gps_poll**(struct gps_data_t *gpsdata)

Arguments
gpsdata – pointer to the GPS data structure used to store GPS information

Return value
0 if successful, -1 otherwise

Example

```
struct gps_data_t *gpsdata;    /* Struct for gps data */

if ( gps_poll(gpsdata) != 0 )
{
  /* Error Management */
}
```

3.3.2.6. gps_del_callback()

gps_del_callback() deregisters the callback function previously set with **gps_set_callback()**. After the invocation of this function no operation will be done when new data arrives.

Prototype

int **gps_del_callback**(struct gps_data_t *gpsdata, pthread_t *handler)

Arguments

gpsdata – pointer to the GPS data structure used to store GPS information
handler – thread handler

Return Value

Same as the return value of pthread_cancel() function:
If successful, the pthread_cancel() function shall return zero; otherwise, an error number shall be returned to indicate the error.

Example

```
struct gps_data_t *gpsdata;    /* Struct for gps data */
void update_gps_panel(struct gps_data_t *gpsdata, char *message);

int main()
{
  pthread_t threadID;
  …..
  gps_set_callback(gpsdata, update_gps_panel, &threadID);
  (void)gps_poll(gpsdata);

  …..
  …..

  If ( gps_del_callback(gpsdata, &threadID) != 0 )
  {
         /* Error Management */
  }
  ….
}
```

3.3.2.7. gps_close()

gps_close() ends the session.

Prototype
int **gps_close**(struct gps_data_t * gpsdata)
Arguments
gpsdata – pointer to the GPS data structure used to store GPS information

Return value
0 if successful, -1 otherwise

Example
```
struct gps_data_t *gpsdata;    /* Struct for gps data */

if( gps_close(gpsdata) != 0 )
{
  /* Error Management */
}
```

3.4. Client Applications

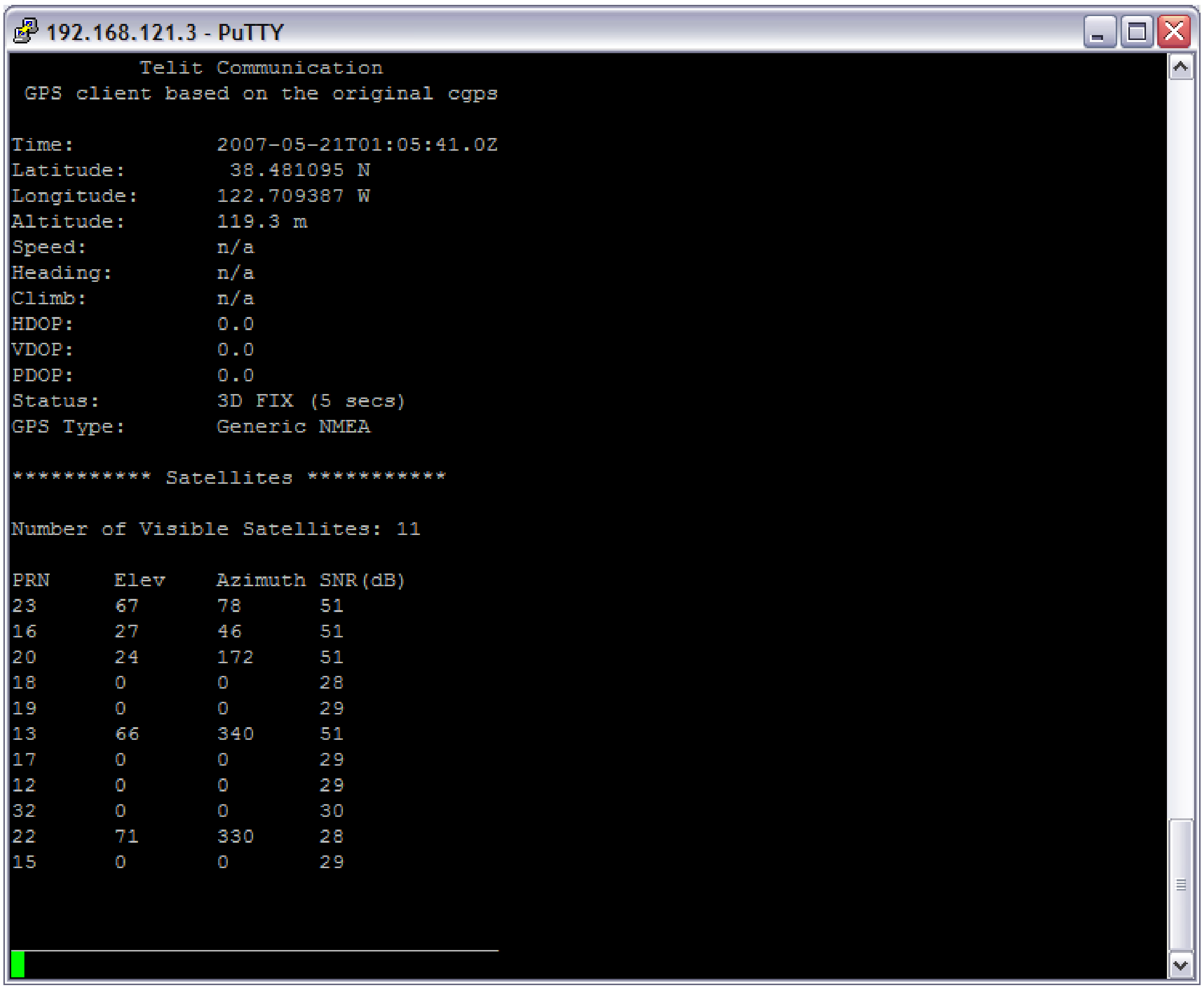
3.4.1. cgps

GE863-Pro^3 GPSD package includes cgps, a simple GPS client application written using libgps APIs that, interfacing with the GPS daemon on TCP port 2947 can show Fix Status, Latitude, Longitude, Altitude, UTC Time, Speed, Heading and Climb Speed. Telit's implementation of cgps for the GE863-Pro^3 is based on the cgps version that can be found into gpsd project's original package.

cgps can be run, once gpsd has be launched, typing:

```
# cgps
```

The screenshot below shows cgps running on GE863-Pro3.



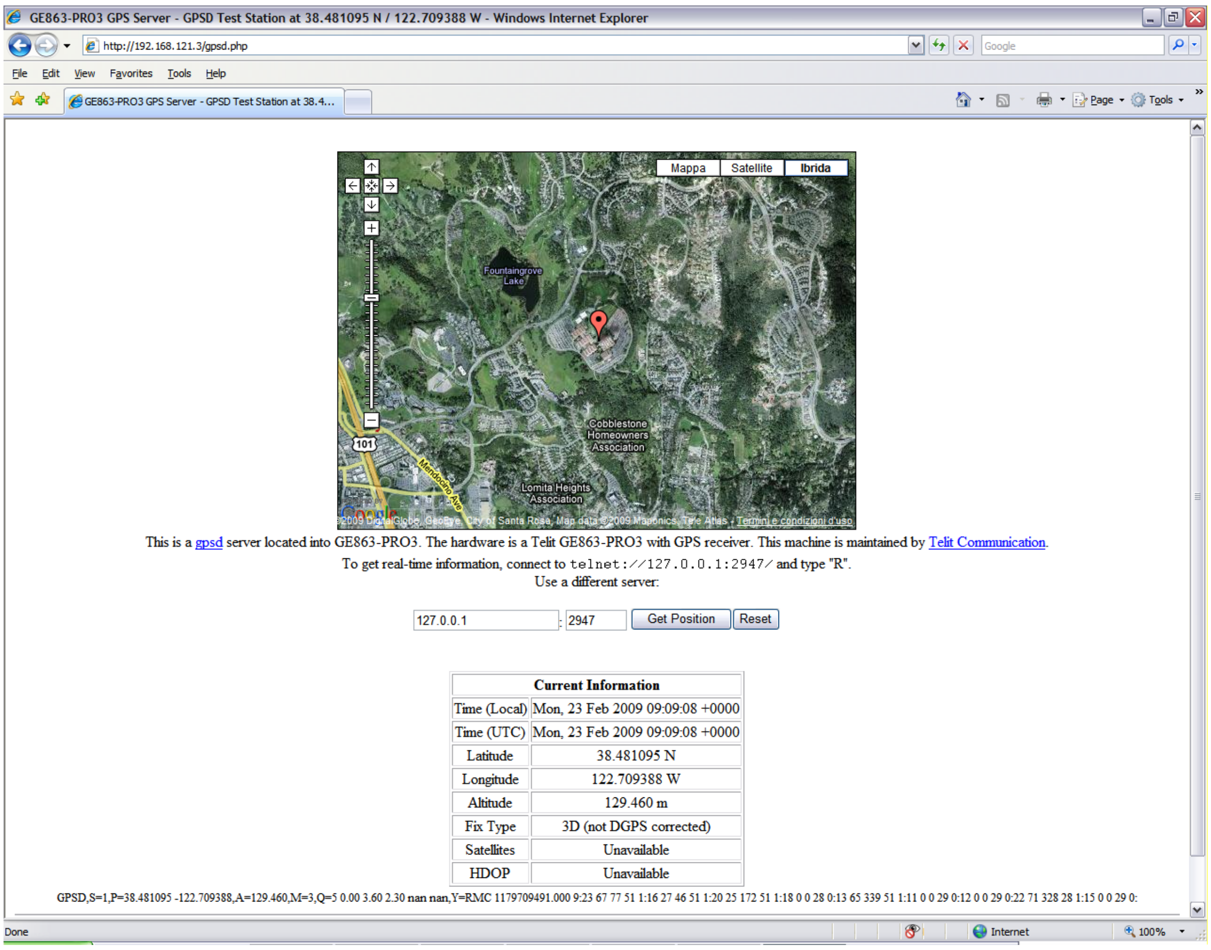
```
          Telit Communication
 GPS client based on the original cgps

Time:           2007-05-21T01:05:41.0Z
Latitude:        38.481095 N
Longitude:      122.709387 W
Altitude:       119.3 m
Speed:          n/a
Heading:        n/a
Climb:          n/a
HDOP:           0.0
VDOP:           0.0
PDOP:           0.0
Status:         3D FIX (5 secs)
GPS Type:       Generic NMEA

*********** Satellites ***********

Number of Visible Satellites: 11

PRN     Elev    Azimuth SNR(dB)
23      67      78      51
16      27      46      51
20      24      172     51
18      0       0       28
19      0       0       29
13      66      340     51
17      0       0       29
12      0       0       29
32      0       0       30
22      71      330     28
15      0       0       29
```

3.4.2. gpsd.php

The GPS package also includes a php script that, interfacing with the GPS daemon on TCP port 2947 and using Google APIs, can show the calculated point onto a web page with Google Maps.
To run gpsd.php test client Cherokee webserver and PHP interpreter must be installed on PRO3.
Also, make sure that your PRO3 EVK is connected to your host PC through the USB Host/Device cable as described in [4].
Open a web browser on your PC and type **http://192.168.121.3/gpsd.php** (192.168.121.3 is the default IP address used by USB Ethernet Gadget Connection [4]).

To use gpsd.php test client with a different IP address the gpsd_config.inc file into /var/www must be edited: a valid google API key must be specified for the new chosen IP address.

The screenshot below shows the gpsd.php example.


GE863-PRO3 GPS Server - GPSD Test Station at 38.481095 N / 122.709388 W - Windows Internet Explorer
http://192.168.121.3/gpsd.php
File Edit View Favorites Tools Help
Mappa
Satellite
Ibrida
Fountaingrove Lake
Cobblestone Homeowners Association
Lomita Heights Association
101
Mendocino Ave
©2009 DigitalGlobe, GeoEye, City of Santa Rosa, Map data ©2009 Maponics, Tele Atlas - Termini e condizioni d'uso
Done
Internet
100%

This is a gpsd server located into GE863-PRO3. The hardware is a Telit GE863-PRO3 with GPS receiver. This machine is maintained by Telit Communication.
To get real-time information, connect to `telnet://127.0.0.1:2947/` and type "R".
Use a different server:

127.0.0.1 : 2947 Get Position Reset

Current Information	
Time (Local)	Mon, 23 Feb 2009 09:09:08 +0000
Time (UTC)	Mon, 23 Feb 2009 09:09:08 +0000
Latitude	38.481095 N
Longitude	122.709388 W
Altitude	129.460 m
Fix Type	3D (not DGPS corrected)
Satellites	Unavailable
HDOP	Unavailable

```
GPSD,S=1,P=38.481095 -122.709388,A=129.460,M=3,Q=5 0.00 3.60 2.30 nan nan,Y=RMC 1179709491.000 9:23 67 77 51 1:16 27 46 51 1:20 25 172 51 1:18 0 0 28 0:13 65 339 51 1:11 0 0 29 0:12 0 0 29 0:22 71 328 28 1:15 0 0 29 0:
```

4. Appendix A

4.1. gpsd clients request/answer protocol

The request protocol for gpsd clients is very simple. Each request normally consists of a single ASCII character followed by a newline. Case of the request character is ignored. Each request returns a line of response text ended by a CR/LF.
Requests and responses, which can be performed through libgps APIs as shown paragraph 3.3, are as follows, with %f standing for a decimal float numeral and %d for decimal integer numeral:

a
The current altitude as "A=%f", meters above mean sea level.

b
The B command with no argument returns four fields giving the parameters of the serial link to the GPS as "B=%d %d %c %d"; baud rate, byte size, parity, (N, O or E for no parity, odd, or even) and stop bits (1 or 2). The command "B=%d" sets the baud rate, not changing parity or stop bits; watch the response, because it is possible for this to fail if the GPS does not support a speed-switching command. In case of failure, the daemon and GPS will continue to communicate at the old speed. The B= form is rejected if more than one client is attached to the channel.

c
C with no following = asks the daemon to return the cycle time of the attached GPS, if any. If there is no attached device it will return "C=?".
If the driver has the capability to change sampling rate the command "C=%f" does so, setting a new cycle time in seconds. The "C=" form is rejected if more than one client is attached to the channel.
If the driver has the capability to change sampling rate, this command always returns "C=%f %f" giving the current cycle time in seconds and the minimum possible cycle time at the current baud rate. If the driver does not have the capability to change sampling rate, this returns, as "C=%f", the cycle time in seconds only.
Either number may be fractional, indicating a GPS cycle shorter than a second; however, if >1 the cycle time must be a whole number. Also note that relatively few GPSes have the ability to set sub-second cycle times; consult your hardware protocol description to make sure this works.
This command will return "C=?" at start of session, before the first full packet has been received from the GPS, because the GPS type is not yet known. To set up conditions for a real answer, issue it after some command that reads position/velocity/time information from the device.

d
Returns the UTC time in the ISO 8601 format "D=yyyy\-mm\-ddThh:nmm:ss\&.ssZ". Digits of precision in the fractional-seconds part will vary and may be absent.

e
Returns "E=%f %f %f": three estimated position errors in meters -- total, horizontal, and vertical (95% confidence level). Note: many GPSes do not supply these numbers. When the GPS does not supply them, gpsd computes them from satellite DOP using fixed figures for expected non-DGPS and DGPS range errors in meters. A value of '?' for any of these numbers should be taken to mean that component of DOP is not available. See also the 'q' command.

f
Gets or sets the active GPS device name. The bare command 'f' requests a response containing 'F=' followed by the name of the active GPS device. The other form of the command is 'f=', in which case all following printable characters up to but not including the next CR/LF are interpreted as the name of a trial GPS device. If the trial device is in gpsd's device list, it is opened and read to see if a GPS can be found there. If it can, the trial device becomes the active device for this client.
The 'f=' command may fail if the specified device name is not on the daemon's device list. This device list is initialized with the paths given on the command line, if any were specified. For security reasons, ordinary clients cannot change this device list; instead, this must be done via the daemon's local control socket declared with the -F option.
Once an 'f=' command succeeds, the client is tied to the specified device until the client disconnects.
Whether the command is 'f' or 'f=' or not, and whether it succeeds or not, the response always lists the name of the client's device.
(At protocol level 1, the F command failed if more than one client was attached, and multiple devices were not supported.)

g
With =, accepts a single argument which may have either of the values 'gps' or 'rtcm104', with case ignored. This specifies the type of information the client wants and forces a device assignment. Without =, forces a device assignment but doesn't force the type. This command is optional; if it is not given, the client will be bound to whatever available device the daemon finds first.
This command returns either '?' if no device of the specified type(s) could be assigned, otherwise a string ('GPS' or 'RTCM104') identifying the kind of information the attached device returns.

i
Returns a text string identifying the GPS. The string may contain spaces and is terminated by CR-LF. This command will return '?' at start of session, before the first full packet has been received from the GPS, because its type is not yet known.

j
Get or set buffering policy; this is only related to NMEA devices which report fix data in several separate sentences during the poll cycle (and in particular it doesn't matter for SiRF chips). The default (j=0) is to clear all fix data at the start of each poll cycle, so until the sentence that reports a given piece of data arrives queries will report ?. Setting j=1 will disable this, retaining data from the previous cycle. This is a per-user-channel bit, not a per\device one. The j=0 setting is hyper-correct and never displays stale data, but may produce a jittery display; the j=1 setting allows stale data but smoothes the display.
(At protocol level below 3, there was no J command. Note, this command is experimental and its semantics are subject to change.)

k
Returns a line consisting of "K=" followed by an integer count of of all GPS devices known to gpsd, followed by a space, followed by a space\-separated list of the device names\&. This command lists devices the daemon has been pointed at by the command\-line argument(s) or an add command via its control socket, and has successfully recognized as GPSes\&. Because GPSes might be unplugged at any time, the presence of a name in this list does not guarantee that the device is available.
(At protocol level 1, there was no K command.)

l
Returns three fields: a protocol revision number, the gpsd version, and a list of accepted request letters.

m
The NMEA mode as "M=%d". 0=no mode value yet seen, 1=no fix, 2=2D (no altitude), 3=3D (with altitude).

n
Get or set the GPS driver mode. Without argument, reports the mode as "N=%d"; N=0 means NMEA mode and N=1 means alternate mode (binary if it has one, for SiRF and Evermore chipsets in particular). With argument, set the mode if possible; the new mode will be reported in the response. The "N=" form is rejected if more than one client is attached to the channel.

o

Attempts to return a complete time/position/velocity report as a unit. Any field for which data is not available being reported as ?. If there is no fix, the response is simply "O=?", otherwise a tag and timestamp are always reported. Fields are as follows, in order:

tag
A tag identifying the last sentence received. For NMEA devices this is just the NMEA sentence name; the talker-ID portion may be useful for distinguishing among results produced by different NMEA talkers in the same wire.

timestamp
Seconds since the Unix epoch, UTC. May have a fractional part of up to .01sec precision.

time error
Estimated timestamp error (%f, seconds, 95% confidence).

latitude
Latitude as in the P report (%f, degrees).

longitude
Longitude as in the P report (%f, degrees).

altitude
Altitude as in the A report (%f, meters). If the mode field is not 3 this is an estimate and should be treated as unreliable.

horizontal error estimate
Horizontal error estimate as in the E report (%f, meters).

vertical error estimate
Vertical error estimate as in the E report (%f, meters).

course over ground
Track as in the T report (%f, degrees).

speed over ground
Speed (%f, meters/sec). Note: older versions of the O command reported this field in knots.

climb/sink
Vertical velocity as in the U report (%f, meters/sec).

estimated error in course over ground
Error estimate for course (%f, degrees, 95% confidence).

estimated error in speed over ground
Error estimate for speed (%f, meters/sec, 95% confidence). Note: older experimental versions of the O command reported this field in knots.

estimated error in climb/sink
Estimated error for climb/sink (%f, meters/sec, 95% confidence).

mode
The NMEA mode (2=2D fix, 3=3D fix). (This field was not reported at protocol levels 2 and lower.)

p
Returns the current position in the form "P=%f %f"; numbers are in degrees, latitude first.

q
Returns "Q=%d %f %f %f %f %f": a count of satellites used in the last fix, and five dimensionless dilution-of-precision (DOP) numbers -- spherical, horizontal, vertical, time, and total geometric. These are computed from the satellite geometry; they are factors by which to multiply the estimated UERE (user error in meters at specified confidence level due to ionosphere's delay, multipath reception, etc.) to get actual circular error ranges in meters (or seconds) at the same confidence level. See also the 'e' command. Note: some GPSes may fail to report these, or report only one of them (often HDOP); a value of 0.0 should be taken as an indication that the data is not available.

r
Sets or toggles 'raw' mode. Return "R=0" or "R=1" or "R=2". In raw mode you read the NMEA data stream from each GPS. (Non-NMEA GPSes get their communication format translated to NMEA on the fly.) If the device is a source of RTCM-104 corrections, the corrections are dumped in the textual format described in rtcm104(5).
The command 'r' immediately followed by the digit '1' or the plus sign '+' sets raw mode. The command 'r' immediately followed by the digit '2' sets super-raw mode; for non-NMEA (binary) GPSes or RTCM-104 sources this dumps the raw binary packet. The command 'r' followed by the digit '0' or the minus sign '-' clears raw mode. The command 'r' with neither suffix toggles raw mode.

s
The NMEA status as "S=%d"\&. 0=no fix, 1=fix, 2=DGPS-corrected fix.

t
Track made good; course "T=%f" in degrees from true north.

u
Current rate of climb as "U=%f" in meters per second. Some GPSes (not SiRF-based) do not report this, in that case gpsd computes it using the altitude from the last fix (if available).

v
The current speed over ground as "V=%f" in knots.

w
Sets or toggles 'watcher' mode (see the description below). Return "W=0" or "W=1".The command 'w' immediately followed by the digit '1' or the plus sign '+' sets watcher mode. The command 'w' followed by the digit '0' or the minus sign '-' clears watcher mode. The command 'w' with neither suffix toggles watcher mode.

x
Returns "X=0" if the GPS is offline, "X=%f" if online; in the latter case, %f is a timestamp from when the last sentence was received.
(At protocol level 1, the nonzero response was always 1.)

y
Returns Y=, followed by a sentence tag, followed by a timestamp (seconds since the Unix epoch, UTC) and a count not more than 12, followed by that many quintuples of satellite PRNs, elevation/azimuth pairs (elevation an integer formatted as %d in range 0-90, azimuth an integer formatted as %d in range 0-359), signal strengths in decibels, and 1 or 0 according as the satellite was or was not used in the last fix. Each number is followed by one space.
(At protocol level 1, this response had no tag or timestamp.)

z
The Z command returns daemon profiling information of interest to gpsd developers. The format of this string is subject to change without notice.

$
The $ command returns daemon profiling information of interest to gpsd developers. The format of this string is subject to change without notice.

Note that a response consisting of just ? following the = means that there is no valid data available. This may mean either that the device being queried is offline, or (for position/velocity/time queries) that it is online but has no fix.
Requests can be concatenated and sent as a string; gpsd will then respond with a comma-separated list of replies.
Every gpsd reply will start with the string "GPSD" followed by the replies. Examples:

query: "p\n" reply: "GPSD,P=36.000000 123.000000\r\n"
query: "d\n" reply: "GPSD,D=2002-11-16T02:45:05.12Z\r\n"
query: "va\n" reply: "GPSD,V=0.000000,A=37.900000\r\n"

When clients are active but the GPS is not responding, gpsd will spin trying to open the GPS device once per second. Thus, it can be left running in background and survive having a GPS repeatedly unplugged and plugged back in. When it is properly installed along with hotplug notifier scripts feeding it device-add commands, gpsd should require no configuration or user action to find devices.
The recommended mode for clients is watcher mode. In watcher mode gpsd ships a line of data to the client each time the GPS gets either a fix update or a satellite picture, but rather than being raw NMEA the line is a gpsd 'o' or 'y' response. Additionally, watching clients get notifications in the form X=0 or X=%f when the online/offline status of the GPS changes, and an I response giving the device type when the user is assigned a device.
Clients should be prepared for the possibility that additional fields (such as heading or roll/pitch/yaw) may be added to the O command, and not treat the occurrence of extra fields as an error. The protocol number will be incremented if and when such fields are added.
Sending SIGHUP to a running gpsd forces it to close all GPSes and all client connections. It will then attempt to reconnect to any GPSes on its device list and resume listening for client connections. This may be useful if your GPS enters a wedged or confused state but can be soft-reset by pulling down DTR.

5. Appendix B

5.1. GPS Acronyms, Abbreviations and Glossary

2D (Two Dimensional)
The horizontal position with latitude/longitude (or northing/easting or X/Y) is called 2D

3D (Three Dimensional)
The horizontal and vertical position with latitude/longitude/altitude (northing/easting/altitude or X/Y/Z) is called 3D coordinate.

AGPS, Assisted GPS
Is a system that enhances the startup performance of a GPS satellite-based positioning system.

Azimuth
A horizontal direction expressed as an angle between a referenced direction, and the direction of the object. The referenced direction is normally true North.

Cold Start
Powering up a unit after it has been turned off for an extended period of time and no longer contains current ephemeris data. In Cold Start Scenario, the receiver has no knowledge on last position, approximate time or satellite constellation. The receiver starts to search for signals blindly. This is normal behavior, if no backup battery is connected. Cold Start time is the longest startup time for GPS receivers and can be several minutes.

Dilution of Precision (DOP)
A description of the purely geometrical contribution to the uncertainty in a position fix.
Standard terms for the GPS application are:
GDOP: Geometric (3 position coordinates plus clock offset in the solution)
PDOP: Position (3 coordinates)
HDOP: Horizontal (2 horizontal coordinates)
VDOP: Vertical (height only)
TDOP: Time (clock offset only)
RDOP: Relative (normalized to 60 seconds and based on a change in geometry)
DOP is a function expressing the mathematical quality of solutions based on the geometry of the satellites. Position dilution of precision (PDOP), the most common such value, has a best case value of 1, higher numbers being worse. A low number of DOP (2) is good, a high number (>7) is considered to be bad. The best PDOP would occur with one satellite directly overhead and three others evenly spaced about the horizon.. PDOP could theoretically be infinite, if all the satellites were in the same plane. PDOP

has a multiplicative effect on the user range error (URE) value. A URE of 32 meters with a PDOP of one would give a user an assumed best accuracy of 32 meters. A PDOP of 2 would result in an assumed accuracy of 64 meters. Many receivers can be programmed to stop providing position solutions above a specific PDOP level (6 is common).

Elevation
Height above or below mean sea level or vertical distance above the geoid.

GNSS - Global Navigation Satellite System
Is the standard generic term for **Satellite Navigation Systems** that provide autonomous geo-spatial positioning with global coverage. As of 2007, the United States NAVSTAR Global Positioning System (GPS) is the only fully operational GNSS.

Hot Start
Start mode of the GPS receiver when current position, clock offset, approximate GPS time and current ephemeris data are all available. In Hot Start Scenario, the receiver was off for less than 2 hours. It uses its last Ephemeris data to calculate a position fix.

HDOP
Horizontal Dilution of Precision. (see Dilution of Precision).

NMEA - National Marine Electronics Association
A U.S. standards committee that defines data message structure, contents, and protocols to allow the GPS receiver to communicate with other pieces of electronic equipment. NMEA 0183 is the standard data communication protocol used by GPS receivers and other types of navigation and marine electronics.

PDOP
Dilution of Precision for a position (3D). (see Dilution of Precision).

RTCM-104
Serial protocol used for broadcasting pseudo range corrections from differential-GPS reference stations.

Time-To-First-Fix (TTFF)
The actual time required by a GPS receiver to achieve a position solution. This specification will vary with the operating state of the receiver, the length of time since the last position fix, the location of the last fix, and the specific receiver design. See also Hot Start, Warm Start and Cold Start mode descriptions.

VDOP
Vertical Dilution of Precision. (see Dilution of Precision).

Warm Start
Start mode of a GPS receiver when current position, clock offset and approximate GPS time are known. Almanac data is retained, but the ephemeris data is cleared. In Warm Start Scenario, the receiver knows - due to a backup battery or by other techniques – his last position, approximate time and almanac. Thanks to this, it can quickly acquire satellites and get a position fix faster than in cold start mode.