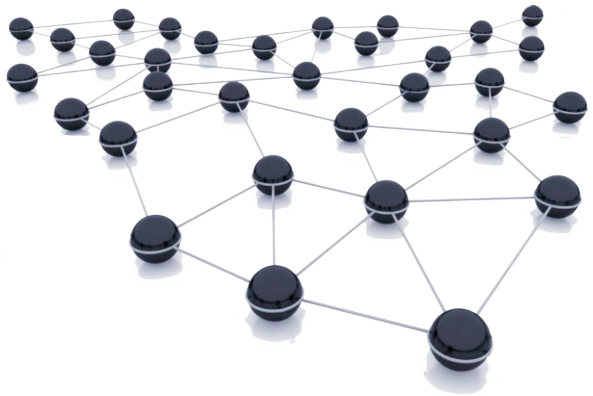
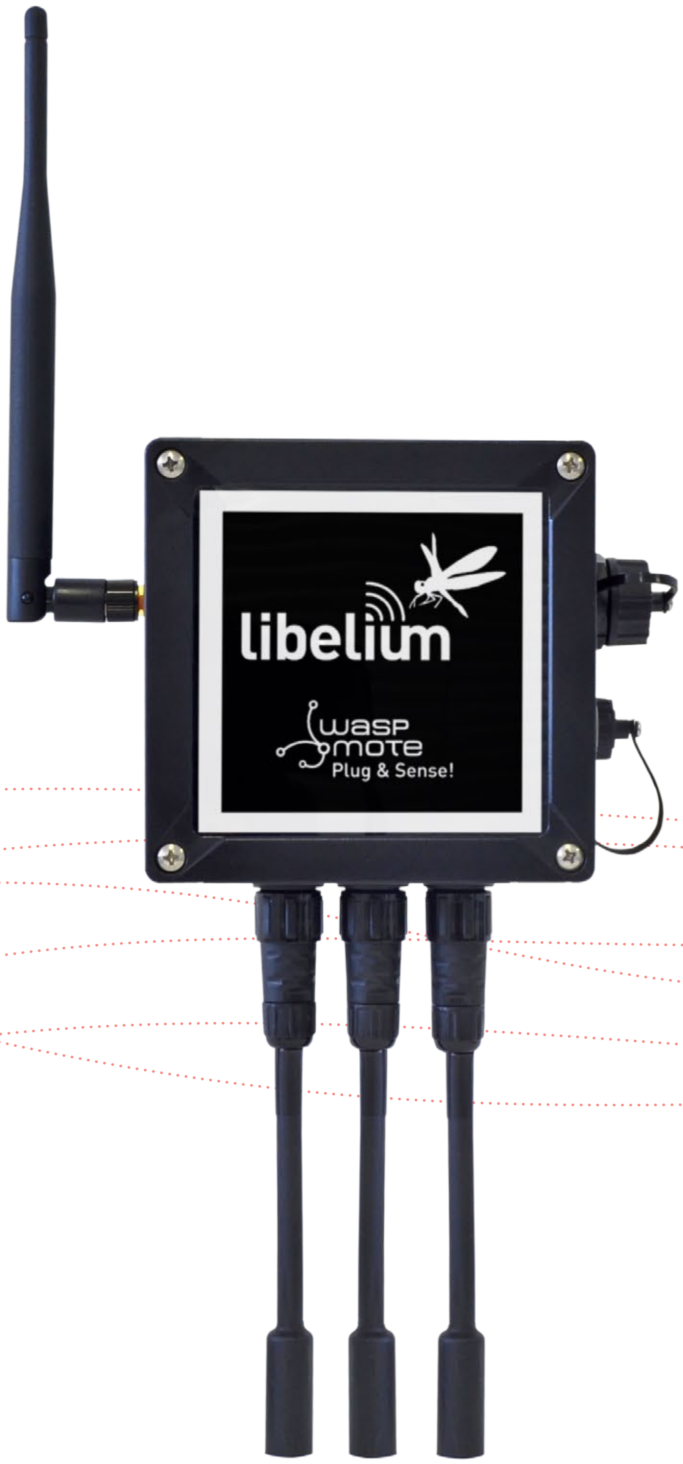
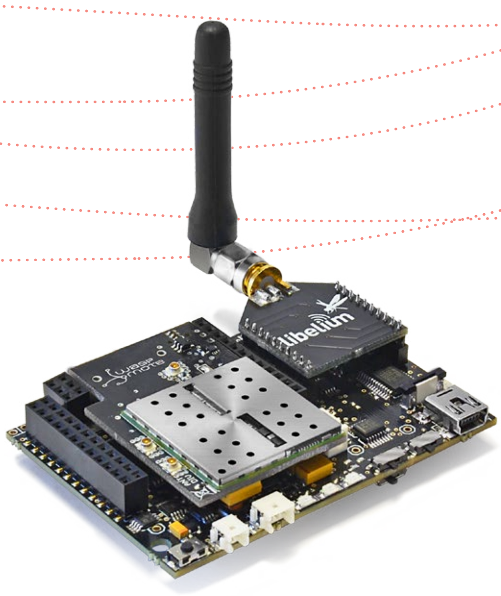
Waspmote GPS

Programming Guide



Document Version: v7.0 - 05/2017

INDEX

1. Introduction

This guide explains the Waspmote GPS module. There are no great variations in this library for our new product lines Waspmote v15 and Plug & Sense! v15, released on October 2016.

Anyway, if you are using previous versions of our products, please use the corresponding guides, available on our Development website. You can get more information about the generation change on the document "New generation of Libelium product lines".

2. Hardware

Waspmote can integrate a GPS receiver which allows to know the exact outside location of the mote anytime. In this way the exact position of the mote can be obtained and even the current time and date, to synchronize the Waspmote internal clock (RTC) with the real time.

- **Model:** JN3 (Telit)
- **Sensitivity:**
  - Acquisition: -147 dBm
  - Navigation: -160 dBm
  - Tracking: -163 dBm
- **Hot start time:** <1 s
- **Cold start time:** <35 s
- **Positional accuracy error:** < 2.5 m
- **Speed accuracy:** < 0.01 m/s
- **EGNOS, WAAS, GAGAN and MSAS capability**

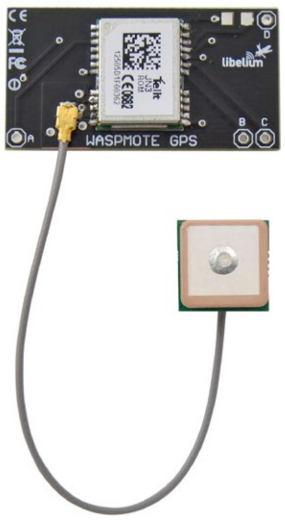


Figure : GPS module

Antenna (Waspmote OEM):
- Cable length: 9 cm
- Connector: UFL
- Gain: 24 dBi (active)

Antenna (Plug & Sense!):
- Cable length: 2 m
- Connector: SMA
- Gain: 26 dBi (active)

Available information: latitude, longitude, altitude, speed, direction, date&time and ephemeris management

Note 2: *The test points named as A, B, C and D are not used to obtain GPS position information and there is no need to use them in any of the features described in this guide. They are just for test purposes and other not relevant features. They correspond with module pinout as follows:*

- A - 1PPS
- B - SCA
- C - SCL
- D - V_BAT

Due to the GPS signal is weak inside buildings, It is recommended to use the GPS module outdoors, with a direct line of sight to the sky. This will ensure the necessary signal quality to obtain valid GPS data.

The GPS module gives us information about:

- latitude
- longitude
- altitude
- speed
- direction
- date/time
- ephemeris

The functions implemented in the API allow this information to be extracted simply, calling functions such as:

```
{
 GPS.getAltitude();
 GPS.getSpeed();
 GPS.getLongitude();
 GPS.getLatitude();
}
```

The GPS receiver uses the UART_1 to communicate with the microcontroller, sharing this UART with the other modules plugged on socket 1. As the 2 modules share this UART, a multiplexer has been enabled in order to select the module with which we wish to communicate at any time. This is not a problem; since all actions are **sequential**, in practice there is **parallel availability** of both devices.

The GPS starts up by default at 4800 bps. This speed can be increased using the library functions that have been designed for controlling and managing the module.

The GPS receiver has 2 operational modes: **NMEA** (National Marine Electronic Association) mode and **binary mode**. NMEA mode uses statements from this standard to obtain `location`, `time` and `date`. The binary mode is based on the sending of structured frames to establish communication between the microcontroller and the GPS receiver, i.e. to read/set ephemeris.

The different types of NMEA statements that the Waspmote's built in GPS receiver supports are:

- NMEA GGA: provides location data and an indicator of data accuracy
- NMEA GSA: provides the status of the satellites the GPS receiver has been connected to
- NMEA GSV: provides information about the satellites the GPS receiver has been connected to
- NMEA RMC: provides information about the date, time, location and speed
- NMEA VTG: provides information about the speed and course of the GPS receiver
- NEMA GLL: provides information about the location of the GPS receiver

The most important NMEA statements are the GGA statements which provide a validity indicator of the measurement carried out, the RMC statement which provides location, speed and date/time and the GSA statement which provides information about the status of the satellites the GPS receiver has been connected to.

(To obtain more information about the NMEA standard and the NMEA statements, visit the website: http://www.gpsinformation.org/dale/nmea.htm)

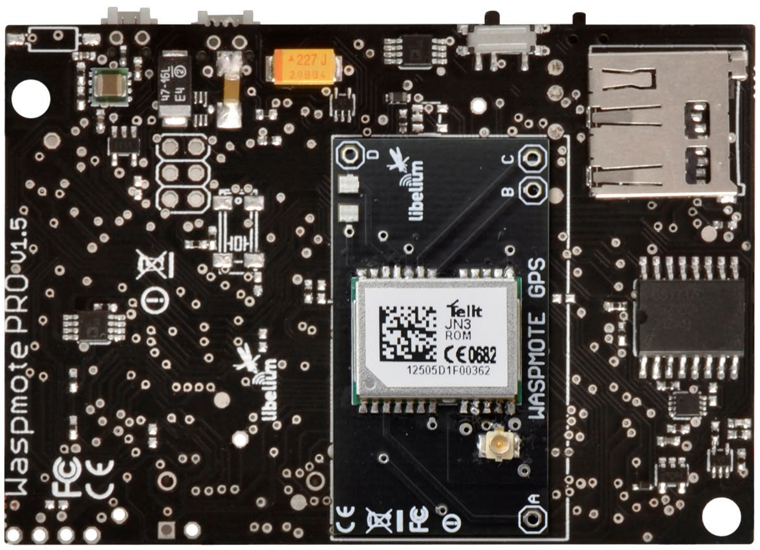


Figure : GPS module connected to Waspmote

3. General considerations

The Waspmote GPS library works with NMEA sentences for getting coordinates. By default, NMEA mode will be used to obtain the GPS information.

3.1. Waspmote libraries

3.1.1. Waspmote GPS files

WaspGPS.h ; WaspGPS.cpp

It is mandatory to include the GPS library when using this module. The following line must be introduced at the beginning of the code:

```
#include <WaspGPS.h>
```

3.1.2. Constructor

To start using the Waspmote GPS library, an object from class `WaspGPS` must be created. This object, called `GPS`, is created inside the Waspmote GPS library and it is public to all libraries. It is used through the guide to show how the Waspmote GPS library works.

When creating this constructor, some variables are defined with a value by default, that can be modified later. These variables are:

- `flag`: init the flag with nothing (ACK value by default)
- `commMode`: shows the communication mode, NMEA by default
- `pwrMode`: powers on GPS module by default

The UART and Baudrate used by default are UART_1 and 4800 bps.

3.1.3. Error messages

When executing some functions, a value is returned explaining the error occurred using the GPS module. The possible values depend on each specific function.

- "error: the GPS timed out" : the GPS timed out while waiting for a string.

3.1.4. GPS communication modes

There are two communication modes: binary (OSP) and NMEA. A variable called `commMode` has been created for indicating the communication mode is being used. The possible values are:

- 0 : OSP_MODE: Binary mode is being used
- 1 : NMEA_Mode: NMEA mode is being used

By default, the GPS module is in NMEA mode, where the default output is as follows:

```
$GPRMC,112443.000,A,4140.8596,N,00053.1827,W,0.00,161.29,220813,,,A*7C
$GPGGA,112444.000,4140.8596,N,00053.1827,W,1,06,2.0,207.8,M,51.6,M,,0000*4A
$GPGSA,A,3,25,12,14,24,02,31,,,,,,,2.6,2.0,1.7*35

$GPRMC,112444.000,A,4140.8596,N,00053.1827,W,0.00,161.29,220813,,,A*7B
$GPGGA,112445.000,4140.8596,N,00053.1827,W,1,06,1.9,207.8,M,51.6,M,,0000*41
$GPGSA,A,3,25,12,14,24,02,31,,,,,,,2.6,1.9,1.7*3F
```

```
$GPRMC,112445.000,A,4140.8596,N,00053.1827,W,0.00,161.29,220813,,,A*7A
$GPGGA,112446.000,4140.8596,N,00053.1827,W,1,06,1.9,207.8,M,51.6,M,,0000*42
$GPGSA,A,3,25,12,14,24,02,31,,,,,,,2.6,1.9,1.7*3F

$GPRMC,112446.000,A,4140.8596,N,00053.1827,W,0.00,161.29,220813,,,A*79
$GPGGA,112447.000,4140.8596,N,00053.1827,W,1,06,1.9,207.8,M,51.6,M,,0000*43
$GPGSA,A,3,25,12,14,24,02,31,,,,,,,2.6,1.9,1.7*3F

$GPRMC,112447.000,A,4140.8596,N,00053.1827,W,0.00,161.29,220813,,,A*78
$GPGGA,112448.000,4140.8596,N,00053.1827,W,1,06,1.9,207.8,M,51.6,M,,0000*4C
$GPGSA,A,3,25,12,14,24,02,31,,,,,,,2.6,1.9,1.7*3F

$GPGSV,3,1,12,25,67,302,51,12,54,047,48,14,47,284,49,24,40,120,41*77
$GPGSV,3,2,12,02,21,087,45,31,13,306,38,29,63,160,,27,59,118,*72
$GPGSV,3,3,12,30,57,284,,15,48,261,,09,47,119,,21,03,157,*78
```

The previous output is repeated every five seconds, where the GPRMC, GPGGA and GPGSA sentences are received four times, while the GPGSV information is received only once.

3.2. GPS start

The GPS module needs to recognize satellites before start working. Connecting to satellites is a process that may take some time, typically less than 35 seconds with good signal quality. This time can be reduced if ephemeris are used. See “Ephemeris handling” section for more information.

To know if the position is fixed or not, the field know as “status” in the GPRMC sentence is used. If this field is ‘A’, the data is valid while if it is ‘V’ the data is still not valid. This process is carried out by the `parseRMC` function, used in `waiForSignal()` function, which is described below.

4. Initialization

4.1. Initializing the GPS module

GPS module is connected to a multiplexer, due to GPRS module and 3G/GPRS module are connected to the same microcontroller UART. To start using GPS module, this multiplexer is switched on and the right combination for GPS is selected.

When the module is switched on, the OSP sentence "PSRF150,1*3E" is received, which means the module is ready.

After the communication with the GPS is available, some initial values are used by the `init()` function, to set the initial position of the GPS module. The default values of these variables are set in the constructor. The default position is set on Zaragoza (Spain) and the user can modify them to set other location.

All the necessary configuration for a right GPS initialization is done by the `ON()` function.

Example of use:

```
{
 GPS.ON(); // Turn it on, open the UART and executes the init process
}
```

4.2. Switching GPS off

It turns GPS off and closes the UART

Example of use:

```
{
  GPS.OFF(); // Turns it off and closes the UART
}
```

4.3. Wait for signal connection

Once the GPS is set on, it is necessary to wait for signal connection. The function in charge of this duty is `waitForSignal()`.

This function waits to connect to satellites for a specific time. The default time to wait for signal connection is 60 seconds. But another time can be specified. If the GPS module connects to satellites, the function returns successfully. If the GPS does not connect to satellites for the specified time, then returns error.

It returns '1' if the GPS has connected within the period of time and '0' if not.

Example of use:

```
{
   // define timeout
   #define GPS_TIMEOUT 200

   // wait to connect for a maximum time of 200 seconds
   bool status = GPS.waitForSignal(GPS_TIMEOUT);
   if(status == true)
   {
      // connected
   }
   else
   {
      // not connected
   }
}
```

4.4. Setting the GPS Power Mode

The function `setMode()` sets the current internal Power Mode on the GPS. The GPS module has two different power modes: `GPS_ON` and `GPS_OFF`. This function is used by default inside `ON()` and `OFF()` functions. The function will set up the `pwrMode` variable to one of the two values.

Example of use:

```
{
   GPS.setMode(GPS_ON); // Set GPS on
   GPS.setMode(GPS_OFF); // Set GPS off
   GPS.getMode(); // Get GPS power mode
}
```

Related variables:

`GPS.pwrMode` → stores GPS power mode

4.5. Setting the UART communication

There are a couple of functions to control the state of the microcontroller's UART. But it is recommended to use both `ON()` and `OFF()` functions to switch on/off the GPS module and open/close the microcontroller's UART at the same time.

4.5.1. Open UART

It opens the UART the GPS module is connected to.

Example of use:

```
{
   GPS.begin(); // open UART
}
```

4.5.2. Close UART

It closes the UART the GPS module is connected to. It implies the disconnection of the internal UART drivers inside the ATMEGA1281 processor.

Example of use:

```
{
   GPS.close(); // close UART
}
```

4.6. Setting the communication mode

Sets the communication mode: binary or NMEA. By default, the GPS module starts using NMEA mode .

The possible values have been defined in previous chapters.

Returns ‘1’ if success and ‘0’ if not.

Example of use:

```
{
   GPS.setCommMode(OSP_MODE); // Sets binary mode
   GPS.getCommMode(); // gets the communication mode
}
```

Related variables:

`GPS.commMode` → stores communication mode established in the GPS module

5. Getting information

The principal purpose of a GPS is to get location, time, date and movement information of the device in which it is plugged.

5.1. Getting time

This function gets the current time on the GPS.

The format of the time is a string like "175625.000" – hhmmss.sss. For example: to store the time 13:30:00, it is stored this way: "133000.000".

The time given by the GPS module is stored in the `timeGPS` variable and it is also returned.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
   char* time;
   time=GPS.getTime(); // Get time given by GPS module
}
```

Related variables:

`GPS.timeGPS` → stores time given by the GPS module

5.2. Getting date

This function gets the current date on the GPS.

The format for the time is a string like "220813" – ddmmyy. For example: to store the date 23rd of March 1994, it is stored this way: "230394".

The date given by the GPS module is stored in the `dateGPS` variable and it is also returned.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
  char* date;
  date=GPS.getDate(); // Get date given by GPS module
}
```

Related variables:

`GPS.dateGPS` → stores date given by the GPS module

5.3. Getting longitude

It gets the longitude from the GPS and returns it.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
 char* longitude;
 longitude=GPS.getLongitude(); // Get longitude from GPS
}
```

Related variables:

`GPS.longitude` → stores longitude given by the GPS module

`GPS.EW_indicator` → stores 'E' for Eastern longitudes and 'W' for Western longitudes

Longitude is measured in longitude and degrees. For example, "01131.0000" could be the value stored by the `GPS.longitude` variable, and it means: longitude 11, degrees 31, 0000 minutes.

5.3.1. Converting to degrees

There is a useful function to convert the longitude given by the GPS module to degrees. This function is called `convert2Degrees()` and returns the longitude represented as a float. Range: 180 to -180.

Example of use:

```
{
   float longitude;
   longitude = GPS.convert2Degrees(GPS.longitude, GPS.EW_indicator);
}
```

5.4. Getting latitude

It gets the latitude from the GPS and returns it.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
 char* latitude;
 latitude=GPS.getLatitude(); // Get latitude from GPS
}
```

Related variables:

`GPS. latitude` → stores latitude given by the GPS module

`GPS.NS_indicator` → stores 'N' for Northern latitudes and 'S' for Southern latitudes

Latitude is measured in latitude and degrees. For example, "4807.0380" could be the value stored by the `GPS.latitude` variable, and it means: latitude 48, degrees 07, 0380 minutes.

5.4.1. Converting to degrees

There is a useful function to convert the latitude given by the GPS module to degrees. This function is called `convert2Degrees()` and returns the latitude represented as a float. Range: 90 to -90.

Example of use:

```
{
   float latitude;
   latitude = GPS.convert2Degrees(GPS.latitude , GPS.NS_indicator);
}
```

5.5. Getting altitude

It gets the altitude in units of meters from the GPS and returns it.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
 char* altitude;
 altitude=GPS.getAltitude(); // Get altitude from GPS
}
```

Related variables:

`GPS. altitude` → stores altitude given by the GPS module in units of meters

As the variable stores the result of an NMEA sentence, the value follows the format according to this standard: Altitude is measured in meters. `GPS.altitude` stores the altitude in meters inside a string.

5.6. Getting speed

It gets the speed from the GPS and returns it.

It stores the final value in the variable `speed` as a string, using units of km/h.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
 char* speed;
 speed=GPS.getSpeed(); // Get speed and course from GPS
}
```

Related variables:

`GPS. speed` → stores speed given by the GPS module

5.7. Getting the course

It gets the course from the GPS and returns it.

It stores the final value in the variable `course` as a string, using units of degrees.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
 char* course;
 course=GPS.getCourse(); // Get course and course from GPS
}
```

Related variables:

`course` → stores course over ground given by the GPS module

5.8. Getting position

It gets latitude, longitude, altitude, speed, course, time and date from the GPS.

It makes a call to the GPGGA and GPRMC sentences to extract the data from the GPS.

It stores the final values in the variables `latitude, longitude, altitude, speed, course, timeGPS` and `dateGPS` as strings.

If this function returns 0, it means that the variable has not been updated due to the information was not available from GPS.

Example of use:

```
{
 GPS.getPosition(); // Get all the coordinates from the GPS
}
```

Related variables:

`GPS. latitude` → stores latitude given by GPS module
`GPS. longitude` → stores longitude given by GPS module
`GPS. altitude` → stores altitude given by GPS module
`GPS. speed` → stores speed given by GPS module
`GPS. course` → stores course given by GPS module
`GPS. timeGPS` → stores time given by GPS module
`GPS. dateGPS` → stores date given by GPS module

Getting GPS data example:
http://www.libelium.com/development/waspmote/examples/gps-01-getting-basic-data

Complete GPS example:
http://www.libelium.com/development/waspmote/examples/gps-04-complete-example

Send GPS data via XBee module example:
http://www.libelium.com/development/waspmote/examples/gps-05-send-gps-xbeedm

GPS tracker using Waspmote:
http://www.libelium.com/development/waspmote/examples/gps-06-waspmote-tracker

5.9. Setting RTC time and date from the GPS

It sets Time and Date to the RTC, getting the data from the GPS.

Inside this function, Time and Date are obtained from the GPS module. ‘

It returns nothing and modifies no flag.

Example of use:

```
{
   GPS.setTimeFromGPS(); // Sets time and date to the RTC from the GPS
}
```

Related variables:

`year`, `month`, `day`, `hour`, `minute`, `second` → stores the time and date set

Setting RTC time from GPS example:
http://www.libelium.com/development/waspmote/examples/gps-07-setting-time-from-gps

5.10. Setting RTC time and date from the GPS

Besides than the described variables and functions, there are other variables, less relevant, which are also stored in RAM memory. These variables are:

`GPS.signalStatus` → status field of RMC sentence
`GPS.satellites` → satellites used in the GGA sentence
`GPS.accuracy` → Horizontal dilution of precision in GGA sentence
`GPS.GSAMode1` → Mode 1 field of GSA sentence
`GPS.GSAMode2` → Mode 2 field of GSA sentence
`GPS.PDOPAccuracy` → Position dilution of precision in GSA sentence
`GPS.HDOPAccuracy` → Horizontal dilution of precision in GSA sentence
`GPS.VDOPAccuracy` → Vertical dilution of precision in GSA sentence
`GPS.satellitesInView` → satellites in view field of GSV sentence

6. Ephemeris handling

Take into account that an SD card is needed for saving ephemeris in Waspmote.

6.1. Loading ephemeris

It loads the ephemeris from the FILE_EPHEMERIS file into the GPS.

It returns '1' if success, '0' if error.

Example of use:

```
{
   int8_t epehemState;
   epehemState=GPS.loadEphems(); // Load "EPHEM.TXT" and send them to GPS module
   epehemState=GPS.loadEphems("EPHEM2.TXT"); // Load ephemeris from this file
}
```

6.2. Saving ephemeris

It saves the ephemeris in the system to the FILE_EPHEMERIS file.

It returns 1 for success, 0 for failure.

Example of use:

```
{
   int8_t ephemState;
   ephemState=GPS.saveEphems(); // Save ephemeris into "EPHEM.TXT"
   ephemState=GPS.saveEphems("EPHEM2.TXT" ); // Save ephemeris to "EPHEM2.TXT"
}
```

Using ephemeris basic example:
http://www.libelium.com/development/waspmote/examples/gps-02-using-ephemeris

Show ephemeris improvement example:
http://www.libelium.com/development/waspmote/examples/gps-03-ephemeris-improvement

7. Advanced functions

Some advanced functions have been developed to treat sentences returned by the GPS module. These functions are used by previously explained functions, to treat NMEA sentences and ephemeris.

In case a developer wants to use these functions, please see "WaspGPS.h" and "WaspGPS.cpp". These functions are:

- `parseRMC()`
- `parseGGA()`
- `parseGSA()`
- `parseGSV()`
- `checksum()`
- `getChecksum()`
- `setChecksum()`

8. Code examples and extended information

In the Waspmote Development section you can find complete examples:
http://www.libelium.com/development/waspmote/examples

Example:

```
#include <WaspGPS.h>

// define GPS timeout when connecting to satellites
// this time is defined in seconds (240sec = 4minutes)
#define TIMEOUT 240

// define status variable for GPS connection
bool status;


void setup()
{
  // Open USB port
  USB.ON();
  USB.println(F("GPS_1 example"));

  // Set GPS ON
  GPS.ON();
}



void loop()
{
  ///////////////////////////////////////////////////
  // 1. wait for GPS signal for specific time
  ///////////////////////////////////////////////////
  status = GPS.waitForSignal(TIMEOUT);

  if( status == true )
  {
    USB.println(F("\n----------------------"));
    USB.println(F("Connected"));
    USB.println(F("----------------------"));
  }
  else
  {
    USB.println(F("\n----------------------"));
    USB.println(F("GPS TIMEOUT. NOT connected"));
    USB.println(F("----------------------"));
  }

  ///////////////////////////////////////////////////
  // 2. if GPS is connected then get position
  ///////////////////////////////////////////////////
  if( status == true )
  {
    USB.println(F("\nGET POSITION:"));

    // getPosition function gets all basic data
    GPS.getPosition();

    // Time
    USB.print(F("Time [hhmmss.sss]: "));
    USB.println(GPS.timeGPS);
```

```
      // Date
      USB.print(F(“Date [ddmmyy]: “));
      USB.println(GPS.dateGPS);

      // Latitude
      USB.print(F(“Latitude [ddmm.mmmm]: “));
      USB.println(GPS.latitude);
      USB.print(F(“North/South indicator: “));
      USB.println(GPS.NS_indicator);

      //Longitude
      USB.print(F(“Longitude [dddmm.mmmm]: “));
      USB.println(GPS.longitude);
      USB.print(F(“East/West indicator: “));
      USB.println(GPS.EW_indicator);

      // Altitude
      USB.print(F(“Altitude [m]: “));
      USB.println(GPS.altitude);

      // Speed
      USB.print(F(“Speed [km/h]: “));
      USB.println(GPS.speed);

      // Course
      USB.print(F(“Course [degrees]: “));
      USB.println(GPS.course);


      USB.println(“\nCONVERSION TO DEGREES (USEFUL FOR INTERNET SEARCH):”);
      USB.print(“Latitude (degrees):”);
      USB.println(GPS.convert2Degrees(GPS.latitude, GPS.NS_indicator));
      USB.print(“Longitude (degrees):”);
      USB.println(GPS.convert2Degrees(GPS.longitude, GPS.EW_indicator));

    }
    delay(5000);


  }
```

9. API changelog

Keep track of the software changes on this link:
www.libelium.com/development/waspmote/documentation/changelog/#GPS

10. Certifications

Libelium offers 2 types of IoT sensor platforms, Waspmote OEM and Plug & Sense!:

- **Waspmote OEM** is intended to be used for research purposes or as part of a major product so it needs final certification on the client side. More info at: www.libelium.com/products/waspmote
- **Plug & Sense!** is the line ready to be used out-of-the-box. It includes market certifications. See below the specific list of regulations passed. More info at: www.libelium.com/products/plug-sense

Besides, Meshlium, our multiprotocol router for the IoT, is also certified with the certifications below. Get more info at:
www.libelium.com/products/meshlium

List of certifications for Plug & Sense! and Meshlium:

- CE (Europe)
- FCC (US)
- IC (Canada)
- ANATEL (Brazil)
- RCM (Australia)
- PTCRB (cellular certification for the US)
- AT&T (cellular certification for the US)

Figure : Certifications of the Plug & Sense! product line

You can find all the certification documents at:
www.libelium.com/certifications