Articles / Internet of Things / Arduino

C++ Arduino FTP WiFi ESP32

ESP32 DIY GPS Tracker

deangi
21 Nov 2023 CPOL 0

DIY GPS Location Tracker

Introduction

This article describes the building of a tracking device using an ESP32 and a GPS module. The GPS module has an antenna which determines location using satellites such as GPS, Glonass, Galileo, and Beidou. An ESP32 supplies power to the module and receives location information from it.

At one minute intervals the location is recorded to a location log file in the flash memory. Each log entry contains a date and time stamp as well as the latitude and longitude and speed the the module is reporting.

Periodically when the tracker is within the range of a pre-programmed WiFi access point, the location log file can be examined or uploaded for storage or further processing.

Background

There are a number of space-based orbital navigation systems available that a suitable receiver can use to determine it's location. The first of these systems was called Navstar or Global Positioning System (GPS). The initial test satellites were launched in 1978. The initial use was directed towards military applications, but in 1983 President Ronald Reagan directed that the system be opened to use for commercial and recreational purposes.

Subsequently the Soviet Union developed and began launching their own system called Glonass in the 1980s. By 1995 the Glonass constellation was complete and available for use.

Subsequent satellite navigation systems were developed by the EU (Galileo) and by China (Beidou). In addition satellites developed and launched by Japan and India have provided enhanced coverage and accuracy for localized areas.

Taken as a whole these systems are now referred to as Global Navigation Satellite Systems or GNSS. The various component systems can be used together by some navigation receivers.

In 2023 there are relatively inexpensive GNSS receivers available that are able to take advantage of multiple signals to provide rapid, reliable, and accurate access to location data wherever satellite signals are available.

GNSS Module

For this project I used a module from Amazon sold as a pack of 2 receiver modules with antennas for less than $16 USD. Although the manufacturer data is not directly available from Amazon, it appears that these are a CASIC Multimode Satellite Navigation Receiver (ATGM3 ???) from Hangzhou Zhongke Microelectronics in China. It's not overly critical as this module supports standard communications (as defined below) by default.

GNSS Module Communications (NMEA)

Because the amount of data produced by a GNSS receiver is somewhat moderate (on the order of 100-200 characters per second ASCII) historically the receivers were connected by an RS232 serial connection - a communications system that has been in used since the 1960's. Baud rates of 4800 or 9600 are fairly commonly found on these modules.

Originally GPS capability was fitted onto large ships and large aircraft to provide a backup determination of location, speed, and direction. By the late 1990's hand-held GPS units became available for purchase in the $100-$200 USD range. An of course today most mobile phones, cars, drones, and even bikes some and scooters have their own GPS receiver.

Due to this history the communications for GNSS have been standardized since the 1992 publication of the NMEA 0183 standard by the National Marine Electronics Association. This defines information sentences that are transmitted by the receiver containing information about the receivers location and other operational parameters such as which satellites are in view of it's antenna and so forth. The current version is 4.00 published in 2008.

NMEA 0183 Sentences

All sentences are essentially lines of ASCII (7 bit) text sent by the receiver. Each sentence begins with a $ character and ends with a <CR><LF>. Each sentence has an 8 bit checksum to provide some confidence in the communications. Each sentence starts with a 3-5 (usually 5) letter code which defines the type of sentence being transmitted. Values are transmitted as numbers or strings of characters, separated by the comma character - they are essentially comma-separated-value (CSV) format when viewed as files. In fact you can import a file of NMEA sentences using Excel as a CSV format document.

Here is an example sentence:

```
$GPRMC,092751.000,A,5321.6802,N,00630.3371,W,0.06,31.66,280511,,,A*45
```

- Begins with the $
- Ends with <CR><LF>
- Sentence type is GPRMC
- Sentence checksum is at the end *45 - meaning 45 hex
- Sentence parameters are comma separated
  - 092751.000 - means the time as 09:27:51.000 seconds UTC0
  - A means the receiver thinks it has a valid position determined from satellites in view
  - 5321.6802,N means the latitude is N53 degrees, 21.6802 minutes
  - 00630.3371,W means the longitude is W006 degrees, 30.3371 minutes
  - 0.06 is the speed (in knots or nautical miles per hour, 1.15078 mph, 1.852 kph)
  - 31.66 is the direction of movement from true north
  - 280511 is the date (28th day of 5th month of 2011

This is the sentence that contains the date and time stamp as well as the latitude and longitude as well as the speed and direction.

Constellation Indicators

The first two letters of a 5 letter sentence (GPRMC above) indicate something about which constellation type was used to determine the location that is being reported. Loosely the letters GP are used for GPS, GL are used for Glonass, BD are used for Beidou, GN is used for locations determined by data from multiple systems.

SO:

- GPRMC means a GPS solution
- GLRMC means a Glonass solution
- BDRMC means a Beidou solution
- GNRMC means a multi-constellation solution

Many other NMEA sentences are available. You can check the documentation of the particular GNSS receiver you have to see which are supported.   You can also check the NMEA 0183 specification to see which sentences are defined (although a particular GNSS receiver may not produce all of them).   The Arduino documents contain a good description of the most common sentences - see this link for details:

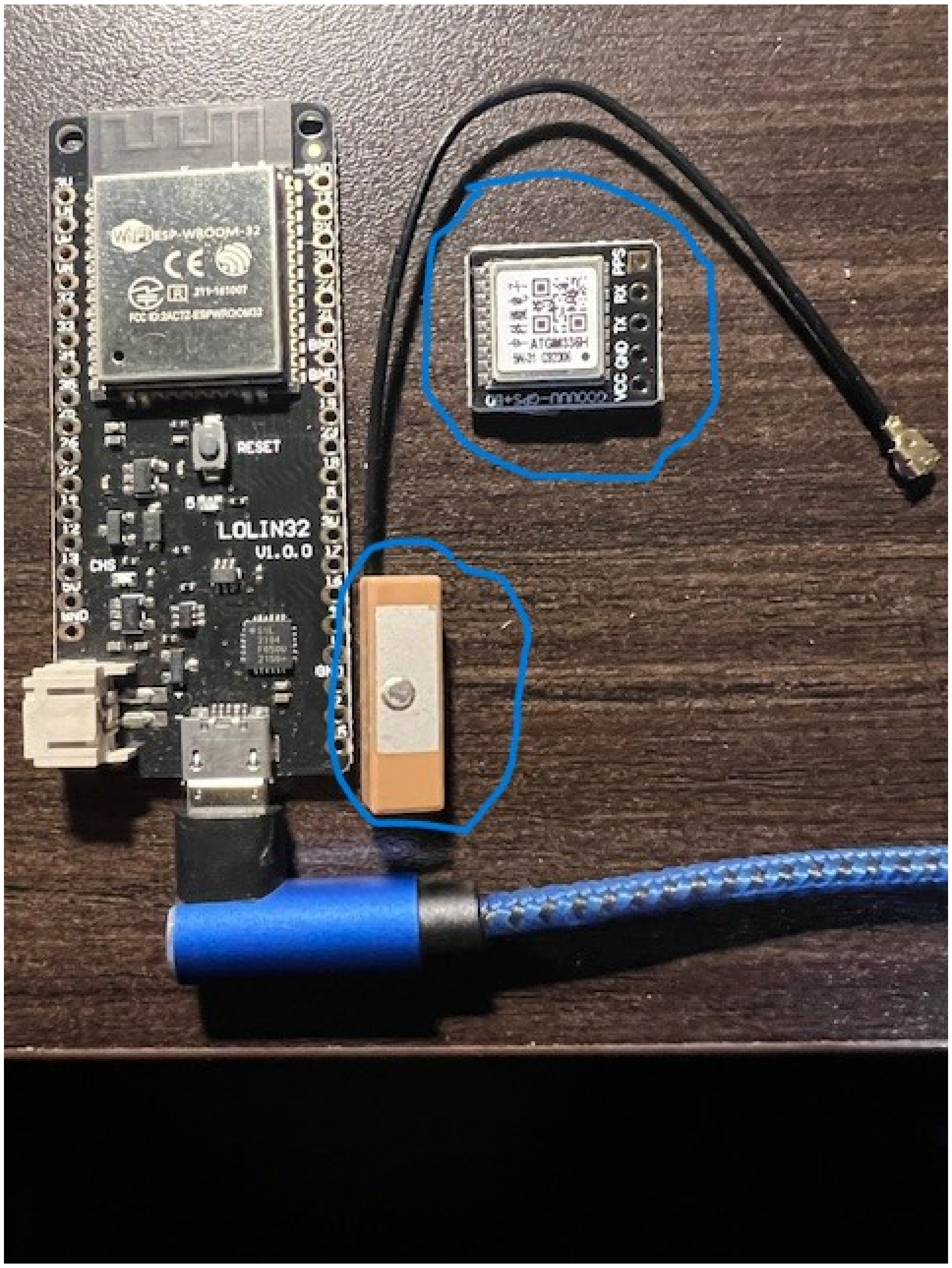
https://docs.arduino.cc/learn/communication/gps-nmea-data-101

Hardware Components

The hardware for this system consists of a two modules.   These can be purchased from Amazon for about $25 to $30 USD.

The processor board is a standard ESP32 board (Search Amazon for ESP WROOM). Many such boards are available on Amazon for $10 USD or less. These boards have built-in WiFi capability as well as Bluetooth.   For this project we will use the following subsystems of the WROOM platform.   However most ESP32 boards will support the required capabilities shown below.

- The processor of course - 32 bit instruction set, floating point, RAM, RTC, Flash, supported by Arduino IDE
- The on-board FLASH and optionally an SD-MMC slot for an SD card
- The WiFi capability
- A serial port to connect to the GNSS module
- A 3.3V (or 5V) output to power the GNSS module



The project will also use a GNSS module which is a small board containing the GNSS receiver itself as well as an internal or external antenna. For this project I ordered a pair of GNSS receiver modules from Amazon:

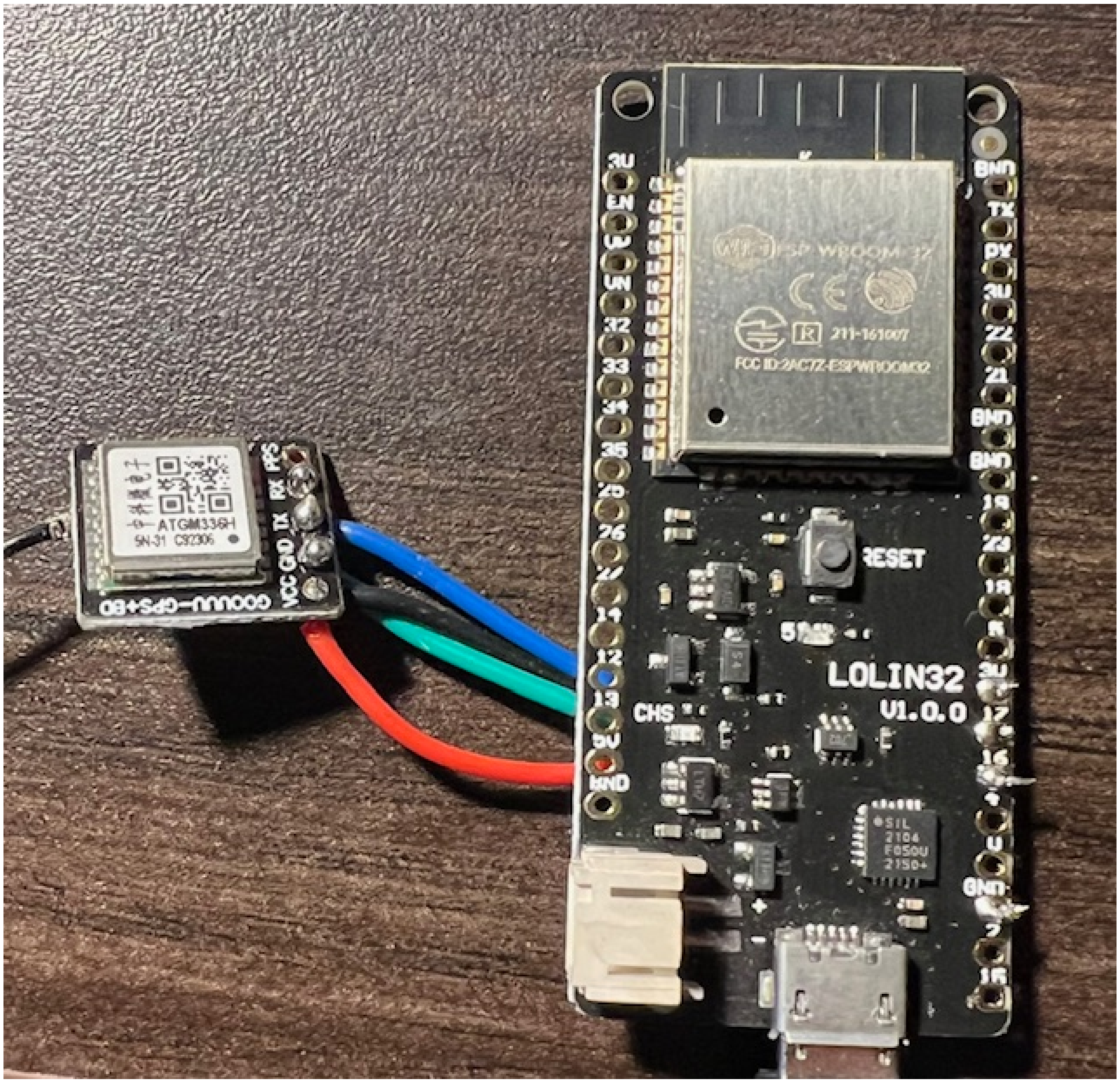
Teyleten Robot ATGM336H GPS+BDS Dual-Mode Module Flight Control Satellite Positioning Navigator

This module runs on 3.3V and produces a TTL-RS232 serial signal at 9600 baud.   We will connect this signal to the ESP32 to receive the NMEA data sent by the module.

There are four wires required to be connected between the ESP32 board and the GNSS module:

- Ground (black) to ESP32 GND

- 3.3 Vdc (red) to ESP32 3.3V
- GNSS TX Data (green) to ESP32 GPIO16 RX Data (ESP Reading data from the module)
- GNSS RX Data (blue) to ESP32 GPIO17 TX Data (ESP Sending data to the module)



Software Code Components

The software was developed using the **Arduino IDE** with the ESP32 board support package. If you're not familiar with this there are many on-line resources on installing the Arduino IDE and the ExpressIF ESP32 board support package.

Make sure to select the following on the Arduino IDE the first time you start it up and open the project.

- Under the tools menu, after you have connected the ESP32 to your PC with the USB cable, select the COM port that communicates to the ESP32 board for programming. For Windows, if you are unsure, you can use the Windows Device Manager to find the COM port.
- Under the tools menu, board selection, under ESP32 Arduino, select the board according to what you are using - probably ESP32 Wroom Module will work if you are unsure.
- Under the tools menu, partition scheme, select Default 4mb with spiffs

Software Components

The following components are used by the software - supplied by the source code or by external libraries provided by the board support package.

- WiFi.h - ESP32 component for basic WiFi support
- WiFiUdp.h - ESP32 component for TCP UDP communications used by NTP
- NTPClient.h - ESP32 component for Network Time Protocol
- SPI.h - ESP32 component for SPI communications
- FS.h - ESP32 file system component
- SPIFFS.h - ESP32 SPI flash file system component
- SD_MMC.h - ESP32 SD card support (if SD card is used)
- ESP32Time.h - ESP32 real-time-clock component
- ESPTelnet.h - ESP32 telnet server component
- ESP32_FTPClient.h - ESP32 FTP client component
- GPS Service - code for servicing the serial port connected to the GNSS module
- FileSystemService - code for supporting reading, writing, creating, deleting, listing files
- Telnet service - code for supporting a simple Telnet shell for accessing the logger
- Logging service - code for logging data to files - including location data from the GNSS module
- Configuration file service - code for reading a configuration file to control WiFi access, FTP access and so forth
- WiFi service - code for connecting/disconnecting/reconnecting to WiFi
- NTP service - code for reading the network time and setting the internal real-time-clock (RTC) (ESP32Time)
- Setup service - code for initial setup of the ESP32 hardware, WiFi, Serial ports, and etc
- Loop service - never ending code for operating the logger - reading and logging GNSS data, handling Telnet, etc.

It seems like a lot, and it is quite a bit - so let's start with a 50,000 foot view and that may clear things up.

50000 foot view

The software will mostly just boot up, initialize everything, the listen for data coming from the GNSS module. Certain GNSS module data (location and time tag) is logged to a file on the board (either spi flash or SD).

To read this data out, a user can connect via Telnet and FTP the logged data to a remote server.

That's it... Now for the details:

Configuration

To operate properly the logger needs to be able to connect to WiFi. It also needs to know what FTP server to use to upload the data and the login credentials for that server.

It also needs to know the time-zone that the logger is operating in, and the baud rate to communicate with the GNSS receiver.

To store these parameters which will vary depending on your implementation and needs, there as a file called config.ini that is stored on the flash file system of the ESP32.

Here is a sample configuration file:

```
// GPS Monitor Config File, Nov 10, 2023, DeanG
WIFISSID=MyWiFi
WIFIPASSWORD=mySecretPassword
TZOFFSETSEC=-28800
FTPSERVER=192.168.4.44
FTPUSER=pi
FTPPASSWORD=raspberry
FTPFOLDER=/media/pi/Seagate2TB/FTP
BAUDRATE=9600
```

This file is stored as config.ini on the flash file system. It will be read on boot-up.

If the config.ini file is not found, the logger can not operate - so make sure it's put there correctly!

Telnet interface

The logger contains a Telnet server and is able to accept Telnet connection requests from external computers over the WiFi interface. When the logger connects to WiFi you will need to check with the WiFi router to see what address is assigned to it. For example, it might be 192.168.5.5. You can also look at the Arduino IDE's Serial Monitor to see the assigned address (Tools, Serial monitor), then reset the ESP32 module and watch for the WiFi connection message that should happen within the first 5-10 seconds after bootup.

For Windows you can use the built-in Telnet client if it is installed. Or you can use the PuTTY program to select a telnet (Other, not SSH) port 22 connection. https://www.putty.org

The telnet interface accepts several simple commands. Don't expect anything like a normal shell here! The file system is flat (no folders) and normally contains only three files;

- config.ini
- location.log
- event.log

The following commands are available:

- ls
- cat /file.1
- cp /file.1 /file.2
- rm /file.1
- on
- off
- ftp

The **ls** command will list the different files on the file system.

The **cat** command will type a file.

The **cp** command will copy a file to a different file.

The **rm** command will remove a file.

The **on** command will turn on logging of the GNSS received data to the telnet interface.

The **off** command will turn off logging of the GNSS received data to the telnet interface.

The **ftp** command will try to send the location.log file to the server defined in the config file.

In normal operation, you would telnet to the logger, then use the ftp command to send the location data to the server.   Once you have verified the information has been correctly received at the server, you would use the rm command to remove the /location.log file.   Removing this file will free up space to be used in future location logging.

Log file

The file system will contain a file called location.log where GNSS information (NMEA $GPRMC sentences) will be recorded each minute of operation.   This results in approximately 4kb per hour of data being logged to the file (< 100kb per day).   For a system with an external SD card, the log file size could exceed 1GB if a large enough card is inserted.   For the internal file system the limit is in the 1.2 to 1.4 MB range - or 12-14 days.   This means the data must be downloaded every week or two with the internal flash file implementation. If the data is stored on an SD card the time between uploads could be much longer.

Flash File System

The flash file system can exist either on the internal SPI flash (which is limited to about 1.4mb size) or it can be stored on a standard SD memory card if the ESP32 you choose supports this. (Some ESP32 boards have an SD card reader socket built-in. Or you can purchase a small circuit board with an SD card reader that can be connected to the ESP32 with a few wires.

Software Startup Procedures

There is a function called setup() which is a standard part of all Arduino sketches in the source code. This function is called one time when the ESP32 boots up. It's task is to configure the hardware modules used by the application and to initialize any software components.

For this application the setup() function needs to:

- Initialize the built-in serial port and print a startup message
- Initialize the built-in ESP32 LED on GPIO2 to use for some comfort signalling
- Start up the flash file system (SD or built in SPI Flash file system - SPIFFS)
- Initialize the real-time clock (it will be set to some default date on boot up, then updated when the NTP service completes)
- Initialize the event logging service (logs some significant events and errors for diagnostic purposes - if everything works right you can ignore this file called event.log)
- Read the configuration file (config.ini)
- Initialize the scheduler service
- Initiate a connection to WiFi
- Initialize the GNSS logging service (location.log)
- Initialize the serial port that receives data from the GNSS module
- Initialize a service that monitors input from the built-in diagnostic port (you can enter shell commands from this port as well as from Telnet)

Here is the setup code:

Shrink ▲

```
//------------------------------------------------------------------------------
// setup() - runs one time when the ESP32 boots up
//------------------------------------------------------------------------------
void setup()
{
  ntpDone = false;
  // Set up the serial port for diagnostic purposes
  Serial.begin(115200);
  // output a signon message to diagnostic port
  Serial.println(SIGNON);

  pinMode(LEDPIN,OUTPUT); // init LED comfort pin

  //----------- initialize the file system ---------------
  // can be either on an SD card or use the built-in flash
  // with the SPI flash file service (SPIFFS)
  // If we can't connect to the file system, the boot-up
  // fails and we can't really go operational.

#ifdef WANTSD_MMC
  // SD card setup
  if(!SD_MMC.begin()) {
    Serial.println("SD Card Mount Failed");
    for (;;);
  }
  uint8_t cardType = SD_MMC.cardType();

  if(cardType == CARD_NONE){
    Serial.println("No SD card attached");
    for (;;);
  }

  Serial.print("SD Card Type: ");
  if(cardType == CARD_MMC){
    Serial.println("MMC");
  } else if(cardType == CARD_SD){
    Serial.println("SDSC");
  } else if(cardType == CARD_SDHC){
    Serial.println("SDHC");
  } else {
    Serial.println("UNKNOWN");
  }

  uint64_t cardSize = SD_MMC.cardSize() / (1024 * 1024);
  Serial.printf("SD Card Size: %lluMB\n", cardSize);
#endif
#ifdef WANTSPIFFS
  // Initialize SPIFFS (file system)
  if(!SPIFFS.begin(true))
  {
    Serial.println("An Error has occurred while mounting SPIFFS");
    return;
  }
  else
  {
    Serial.println("SPIFFS mounted");
  }
#endif

  // initialize the RTC, uses timer 0
  rtc.setTime(00,00,00, 1, 1, 2023); // default time 00:00:00 1/1/2023

  logInit(EVENTFN, true);

  listDir(fileSystem, "/", 1); // for dev purposes, show the file system on boot up

  // read config file
  if (!readConfigFile(CONFIGFN))
  {
    logMessage("Unable to read config file");
  }

  schedulerInit(); // initialize the scheduler used by the loop() function
  // initiate a WIFI connect
  wifiConnect();

  gpsLogInit();

  // Set up serial port for connection to GPS module
  gpsInit(baudRate);

  rmcbuf[0] = '\0';

  setupTelnetDone = false;
  sioInit();  // diagnostic serial port input service

  //log_d("Total heap: %d", ESP.getHeapSize());
  //log_d("Free heap: %d", ESP.getFreeHeap());
  //("Total PSRAM: %d", ESP.getPsramSize());
  //log_d("Free PSRAM: %d", ESP.getFreePsram());

  Serial.print("\n>");  // initial serial prompt
}
```

Software Operational Mode

The operational mode of the software happens all inside a standard Arduino function called loop(). After the setup() function is called in the boot-up process, the loop() function is called repeatedly over and over forever. It is in this function that the main operation mode of the application is implemented.

For this application, a simple scheduler was implemented using the RTC to divide different activities that need to be performed into groups by how frequently the activity needs to be performed. This allows us to have some activities that are done once per second, some that are done once per minute, some once per hour, and some once per day.

In addition, all the remaining time of the ESP32 is devoted to doing "high-rate" activities or tasks such as servicing Telnet, reading data from the GNSS module and so forth.

You'll see in the code below these various tasks divided according to frequency.

- High rate tasks
  - gpsService() - read any characters from the GNSS module. If an entire line is available then process that line
  - telnet.loop() - process any telnet data that may arrive
- Per second tasks
  - wifiService() - handle logic associated with connecting, disconnecting, and reconnecting to WiFi with appropriate time-outs, and, after first WiFi connect, start an NTP request to see what time it is. Also start the Telnet service after WiFi is connected.
  - ntpService() - handle logic associated with communicating over WiFi with an NTP server to get the network time and update the real time clock

    - sioService() - handle any characters typed at the diagnostic serial port - for diag purposes, shell commands can be entered this way
- Per minute tasks
    - Once per minute, log the latest received GNSS location data to the location.log file
- Per hour tasks
    - Once per hour, flush GNSS location data to the location.log file
- Per day tasks
    - Once per day, start a new NTP request to make sure the real time clock remains up-to-date

Here is the source code for the loop() function

Shrink ▲

```
//----------------------------------------------------------------------------
// Main repeatitive tasks go here.  This is called over and over endlessly
// once setup() has completed.
//----------------------------------------------------------------------------
void loop()
{
  // This is sort of a poor-person's operating system - scheduling tasks
  // at periodic intervals.

  //----------------------------
  // high rate tasks here
  //----------------------------
  char* line = gpsService();
  if (line != NULL)
  {
    if (gpsSerialEcho) Serial.println(line);
    if (gpsTelnetEcho && telnetConnected) telnet.println(line);
    // $GxRMC
    //Serial.print(line[0]); Serial.print(line[1]); Serial.print(line[3]);
Serial.print(line[4]); Serial.println(line[5]);
    if ((line[1] == 'G') &&
        (line[3] == 'R') &&
        (line[4] == 'M') &&
        (line[5] == 'C'))  strcpy(rmcbuf,line); // save for minute by minute logging
  }
  telnet.loop(); // process any telnet traffic

  //----------------------------
  // tasks executed once per second
  //----------------------------
  if (secondDetector())
  {
    wifiService(); // service the wifi connection controller

    if (wifiIsConnected() && !ntpStarted() && (ntpAttempts == 0)) ntpStart(); // first NTP
request
    ntpService();
    if (!ntpDone && ntpComplete())
    {
      logMessage("NTP (bootup) completed");
      ntpDone = true;
    }

    if (wifiIsConnected() && !setupTelnetDone)
    {
      setupTelnetDone = true;
      setupTelnet();
    }

    char* sioinputline = sioService();
    if (sioinputline != NULL) handleShellCommand(String(sioinputline));
  }

  //-----------------------------
  // tasks executed once per minute
  //-----------------------------
  if (minuteDetector())
  {
    gpsLogLine(rmcbuf); // log position if available once per minute
  }

  //-----------------------------
  // tasks executed once per hour
  //-----------------------------
  if (hourDetector())
  {
    gpsLogFlush(); // flush log hourly
  }

  //-----------------------------
  // tasks executed once per day
  //-----------------------------
  if (dayDetector())
  {
    ntpStart(); // dayly, get an NTP update
  }
}
```

That's about it for the top-level view of the software.

The following files are part of the source code:

- gpsLogger.ino - source code for the main application
- FileSystemService.h - source code for file-system operations such as reading, writing files
- NTPService.h - source code for sequencing an NTP client request to get the time-of-day
- WiFiService.h - source code for connecting, disconnecting, and reconnecting with a WiFi AP
- sioService.h - source code for reading lines of data from the diagnostic serial port
- SchedulerService.h - source code for dividing up tasks into per-second, per-minute, and so forth

Hope you have fun with this - there's many more ideas that could be added to the software to enhance it for a particular set of application requirements.

History

Version 1.2, November, 2023 - initial operational version.

License

Written By

deangi

Team Leader

United States

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.

Comments and Discussions