DE2-Project-GPS-Tracker

3.0

Generated by Doxygen 1.12.0

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 libGPS Namespace Reference

Data Structures

- class MicropyGPS

5.2 Main_v2 Namespace Reference

Functions

- initialize_display ()
- update_display (oled, gps_data)
- parse_gps_data (my_gps)
- main ()

Variables

- int I2C_SCL_PIN = 22
- int I2C_SDA_PIN = 21
- int I2C_FREQ = 100_000
- int UART_TX_PIN = 17
- int UART_RX_PIN = 16
- int UART_BAUDRATE = 9600
- int OLED_CONTRAST = 80
- int DISPLAY_REFRESH_INTERVAL = 1

5.2.1 Detailed Description

```
================================================================================
Project Name: GPS Data Display on OLED
File Name: Main_v2.0.py
Description: This script reads GPS data from a UART-connected GPS module, parses
             the data using MicropyGPS, and displays it on an SH1106 OLED screen.
             The displayed information includes time, date, latitude, longitude,
             altitude, number of satellites, and horizontal dilution of precision (HDOP).

Authors:
  - Guilherme Brito
  - Henrique Silva
  - João Santos

Version: 2.0
Date Created: 14-11-2024
Last Modified: 21-11-2024

Parameters:
  - I2C Pins: SCL = GP22, SDA = GP21
  - UART Pins: TX = GP17, RX = GP16
  - OLED Contrast: 80%
  - GPS Baud Rate: 9600
  - Display Refresh Interval: 1 second

Usage:
  - Ensure the SH1106 OLED display and GPS module are connected as per the pin
    configuration.
  - Upload the script to a microcontroller and monitor the OLED for GPS data.
  - Exit the program with Ctrl+C.

Dependencies:
  - machine (for I2C and UART)
  - sh1106 (OLED display driver)
  - libGPS (Adaptation of the MicropyGPS for GPS parsing)

Notes:
  - This code is optimize for ESP32
  - See documentation for connections between the three

================================================================================
```

5.2.2 Function Documentation

5.2.2.1 initialize_display()

```
initialize_display ()
```

```
Set up the I2C bus and OLED display.
```

Definition at line 64 of file Main_v2.0.py.

5.2.2.2 main()

```
main ()
```

```
The main program loop. Initializes the OLED display and GPS module, then continuously
updates the display with parsed GPS data.
```

Definition at line 134 of file Main_v2.0.py.

5.2.2.3 parse_gps_data()

```
parse_gps_data (
              my_gps)
```

```
Extracts GPS data from the MicropyGPS object and formats it.
```

Definition at line 115 of file Main_v2.0.py.

5.2.2.4 update_display()

```
update_display (
              oled,
              gps_data)
```

```
Refreshes the OLED display with new GPS data.
```

Definition at line 84 of file Main_v2.0.py.

5.2.3 Variable Documentation

5.2.3.1 DISPLAY_REFRESH_INTERVAL

```
int DISPLAY_REFRESH_INTERVAL = 1
```

Definition at line 58 of file Main_v2.0.py.

5.2.3.2 I2C_FREQ

```
int I2C_FREQ = 100_000
```

Definition at line 53 of file Main_v2.0.py.

5.2.3.3 I2C_SCL_PIN

```
int I2C_SCL_PIN = 22
```

Definition at line 51 of file Main_v2.0.py.

5.2.3.4 I2C_SDA_PIN

```
int I2C_SDA_PIN = 21
```

Definition at line 52 of file Main_v2.0.py.

5.2.3.5 OLED_CONTRAST

```
int OLED_CONTRAST = 80
```

Definition at line 57 of file Main_v2.0.py.

5.2.3.6 UART_BAUDRATE

```
int UART_BAUDRATE = 9600
```

Definition at line 56 of file Main_v2.0.py.

5.2.3.7 UART_RX_PIN

```
int UART_RX_PIN = 16
```

Definition at line 55 of file Main_v2.0.py.

5.2.3.8 UART_TX_PIN

```
int UART_TX_PIN = 17
```

Definition at line 54 of file Main_v2.0.py.

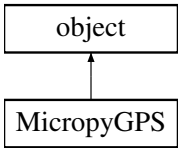
5.3 test Namespace Reference

Chapter 6

Data Structure Documentation

6.1 MicropyGPS Class Reference

Inheritance diagram for MicropyGPS:



Public Member Functions

- __init__ (self, local_offset=0, location_formatting='dd')
- latitude (self)

  Coordinates Translation Functions.
- longitude (self)
- start_logging (self, target_file, mode="append")

  Logging Related Functions.
- stop_logging (self)
- write_log (self, log_string)
- gprmc (self)

  Sentence Parsers.
- gpgll (self)
- gpvtg (self)
- gpgga (self)
- gpgsa (self)
- gpgsv (self)
- new_sentence (self)

  Data Stream Handler Functions.
- update (self, new_char)
- new_fix_time (self)
- satellite_data_updated (self)

  User Helper Functions working with the GPS object data easier.
- unset_satellite_data_updated (self)
- satellites_visible (self)

- time_since_fix (self)
- compass_direction (self)
- latitude_string (self)
- longitude_string (self)
- speed_string (self, unit='kph')
- date_string (self, formatting='s_dmy', century='20')

Data Fields

- bool sentence_active = False
- int active_segment = 0
- bool process_crc = False
- list gps_segments = [ ]
- int crc_xor = 0
- int char_count = 0
- int fix_time = 0
- int crc_fails = 0
- int clean_sentences = 0
- int parsed_sentences = 0
- log_handle = None
- bool log_en = False
- list timestamp = [0,0,0.0]
- list date = [0, 0, 0]
- local_offset = local_offset
- str coord_format = location_formatting
- list speed = [0.0, 0.0, 0.0]
- float course = 0.0
- float altitude = 0.0
- float geoid_height = 0.0
- int satellites_in_view = 0
- int satellites_in_use = 0
- list satellites_used = [ ]
- int last_sv_sentence = 0
- int total_sv_sentences = 0
- satellite_data = dict()
- float hdop = 0.0
- float pdop = 0.0
- float vdop = 0.0
- bool valid = False
- int fix_stat = 0
- int fix_type = 1
- tuple date = 'A':
- str log_en = '$':
- str sentence_active = '*':
- int process_crc = 2:
- bool crc_xor = True
- int supported_sentences = 1

Static Public Attributes

- int SENTENCE_LIMIT = 90
- dict supported_sentences

Protected Attributes

- list _latitude = [0, 0.0, 'N']
- list _longitude = [0, 0.0, 'W']

6.1.1 Detailed Description

```
GPS NMEA Sentence Parser. Creates object that stores all relevant GPS data and statistics.
Parses sentences one character at a time using update().
```

Definition at line 13 of file libGPS.py.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 __init__()

```
__init__ (
              self,
              local_offset = 0,
              location_formatting = 'dd')


Setup GPS Object Status Flags, Internal Data Registers, etc
    local_offset (int): Timzone Difference to UTC
    location_formatting (str): Style For Presenting Longitude/Latitude:
                               Decimal Degree Minute (ddm) - 40° 26.767 N
                               Degrees Minutes Seconds (dms) - 40° 26 46 N
                               Decimal Degrees (dd) - 40.446° N
```

Definition at line 29 of file libGPS.py.

6.1.3 Member Function Documentation

6.1.3.1 compass_direction()

```
compass_direction (
              self)


Determine a cardinal or inter-cardinal direction based on current course.:return: string
```

Definition at line 644 of file libGPS.py.

6.1.3.2 date_string()

```
date_string (
              self,
              formatting = 's_dmy',
              century = '20')
```

```
Creates a readable string of the current date.
Can select between long format: Januray 1st, 2014
or two short formats:
11/01/2014 (MM/DD/YYYY)
01/11/2014 (DD/MM/YYYY)
:param formatting: string 's_mdy', 's_dmy', or 'long'
:param century: int delineating the century the GPS data is from (19 for 19XX, 20 for 20XX)
:return: date_string  string with long or short format date
```

Definition at line 697 of file libGPS.py.

6.1.3.3 gpgga()

```
gpgga (
              self)
```

```
Parse Global Positioning System Fix Data (GGA) Sentence. Updates UTC timestamp, latitude, longitude,
fix status, satellites in use, Horizontal Dilution of Precision (HDOP), altitude, geoid height and fix status
```

Definition at line 318 of file libGPS.py.

6.1.3.4 gpgll()

```
gpgll (
              self)
```

```
Parse Geographic Latitude and Longitude (GLL)Sentence. Updates UTC timestamp, latitude, longitude, and fix sta
```

Definition at line 247 of file libGPS.py.

6.1.3.5 gpgsa()

```
gpgsa (
              self)
```

```
Parse GNSS DOP and Active Satellites (GSA) sentence. Updates GPS fix type, list of satellites used in
fix calculation, Position Dilution of Precision (PDOP), Horizontal Dilution of Precision (HDOP), Vertical
Dilution of Precision (VDOP), and fix status
```

Definition at line 402 of file libGPS.py.

6.1.3.6 gpgsv()

```
gpgsv (
              self)
```

```
Parse Satellites in View (GSV) sentence. Updates number of SV Sentences,the number of the last SV sentence
parsed, and data on each satellite present in the sentence
```

Definition at line 448 of file libGPS.py.

6.1.3.7 gprmc()

```
gprmc (
              self)
```

Sentence Parsers.

```
Parse Recommended Minimum Specific GPS/Transit data (RMC)Sentence.
Updates UTC timestamp, latitude, longitude, Course, Speed, Date, and fix status
```

Definition at line 151 of file libGPS.py.

6.1.3.8 gpvtg()

```
gpvtg (
              self)
```

```
Parse Track Made Good and Ground Speed (VTG) Sentence. Updates speed and course
```

Definition at line 305 of file libGPS.py.

6.1.3.9 latitude()

```
latitude (
              self)
```

Coordinates Translation Functions.

```
Format Latitude Data Correctly
```

Definition at line 89 of file libGPS.py.

6.1.3.10 latitude_string()

```
latitude_string (
              self)
```

```
Create a readable string of the current latitude data:return: string
```

Definition at line 659 of file libGPS.py.

6.1.3.11 longitude()

```
longitude (
              self)
```

```
Format Longitude Data Correctly
```

Definition at line 102 of file libGPS.py.

6.1.3.12 longitude_string()

```
longitude_string (
              self)
```

```
Create a readable string of the current longitude data: return: string
```

Definition at line 671 of file libGPS.py.

6.1.3.13 new_fix_time()

```
new_fix_time (
              self)
```

```
Updates a high resolution counter with current time when fix is updated. Currently only triggered from
GGA, GSA and RMC sentences
```

Definition at line 604 of file libGPS.py.

6.1.3.14 new_sentence()

```
new_sentence (
              self)
```

Data Stream Handler Functions.

```
Adjust Object Flags in Preparation for a New Sentence
```

Definition at line 516 of file libGPS.py.

6.1.3.15 satellite_data_updated()

```
satellite_data_updated (
              self)
```

User Helper Functions working with the GPS object data easier.

```
Checks if the all the GSV sentences in a group have been read, making satellite data complete:return: boolean
```

Definition at line 614 of file libGPS.py.

6.1.3.16 satellites_visible()

```
satellites_visible (
              self)
```

```
Returns a list of of the satellite PRNs currently visible to the receiver:return: list
```

Definition at line 625 of file libGPS.py.

6.1.3.17 speed_string()

```
speed_string (
              self,
              unit = 'kph')
```

```
Creates a readable string of the current speed data in one of three units
:param unit: string of 'kph' or 'mph
:return:
```

Definition at line 683 of file libGPS.py.

6.1.3.18 start_logging()

```
start_logging (
              self,
              target_file,
              mode = "append")
```

Logging Related Functions.

```
Create GPS data log object
```

Definition at line 116 of file libGPS.py.

6.1.3.19 stop_logging()

```
stop_logging (
              self)
```

```
Closes the log file handler and disables further logging
```

Definition at line 130 of file libGPS.py.

6.1.3.20 time_since_fix()

```
time_since_fix (
              self)
```

```
Returns number of millisecond since the last sentence with a valid fix was parsed. Returns 0 if no fix has bee
```

Definition at line 629 of file libGPS.py.

6.1.3.21 unset_satellite_data_updated()

```
unset_satellite_data_updated (
              self)
```

```
Mark GSV sentences as read indicating the data has been used and future updates are fresh
```

Definition at line 621 of file libGPS.py.

6.1.3.22 update()

```
update (
              self,
              new_char)
```

```
Process a new input char and updates GPS object if necessary based on special characters ('$', ',', '*')
Function builds a list of received string that are validate by CRC prior to parsing by the  appropriate
sentence function. Returns sentence type on successful parse, None otherwise
```

Definition at line 525 of file libGPS.py.

6.1.3.23 write_log()

```
write_log (
              self,
              log_string)
```

```
Attempts to write the last valid NMEA sentence character to the active file handler
```

Definition at line 141 of file libGPS.py.

6.1.4 Field Documentation

6.1.4.1 _latitude

```
list _latitude = [0, 0.0, 'N']  [protected]
```

Definition at line 64 of file libGPS.py.

6.1.4.2 _longitude

```
list _longitude = [0, 0.0, 'W']  [protected]
```

Definition at line 65 of file libGPS.py.

6.1.4.3 active_segment

```
int active_segment = 0
```

Definition at line 41 of file libGPS.py.

6.1.4.4 altitude

```
float altitude = 0.0
```

Definition at line 69 of file libGPS.py.

6.1.4.5 char_count

```
int char_count = 0
```

Definition at line 45 of file libGPS.py.

6.1.4.6 clean_sentences

```
int clean_sentences = 0
```

Definition at line 50 of file libGPS.py.

6.1.4.7 coord_format

```
str coord_format = location_formatting
```

Definition at line 66 of file libGPS.py.

6.1.4.8 course

```
float course = 0.0
```

Definition at line 68 of file libGPS.py.

6.1.4.9 crc_fails

```
int crc_fails = 0
```

Definition at line 49 of file libGPS.py.

6.1.4.10 crc_xor [1/2]

```
int crc_xor = 0
```

Definition at line 44 of file libGPS.py.

6.1.4.11 crc_xor [2/2]

```
bool crc_xor = True
```

Definition at line 572 of file libGPS.py.

6.1.4.12 date [1/2]

```
list date = [0, 0, 0]
```

Definition at line 60 of file libGPS.py.

6.1.4.13 date [2/2]

```
tuple date = 'A':
```

Definition at line 182 of file libGPS.py.

6.1.4.14 fix_stat

```
int fix_stat = 0
```

Definition at line 83 of file libGPS.py.

6.1.4.15 fix_time

```
int fix_time = 0
```

Definition at line 46 of file libGPS.py.

6.1.4.16 fix_type

```
int fix_type = 1
```

Definition at line 84 of file libGPS.py.

6.1.4.17 geoid_height

```
float geoid_height = 0.0
```

Definition at line 70 of file libGPS.py.

6.1.4.18 gps_segments

```
list gps_segments = []
```

Definition at line 43 of file libGPS.py.

6.1.4.19 hdop

```
float hdop = 0.0
```

Definition at line 79 of file libGPS.py.

6.1.4.20 last_sv_sentence

```
int last_sv_sentence = 0
```

Definition at line 76 of file libGPS.py.

6.1.4.21 local_offset

```
local_offset = local_offset
```

Definition at line 61 of file libGPS.py.

6.1.4.22 log_en [1/2]

```
bool log_en = False
```

Definition at line 55 of file libGPS.py.

6.1.4.23 log_en [2/2]

```
str log_en = '$':
```

Definition at line 539 of file libGPS.py.

6.1.4.24 log_handle

```
log_handle = None
```

Definition at line 54 of file libGPS.py.

6.1.4.25 parsed_sentences

```
int parsed_sentences = 0
```

Definition at line 51 of file libGPS.py.

6.1.4.26 pdop

```
float pdop = 0.0
```

Definition at line 80 of file libGPS.py.

6.1.4.27 process_crc [1/2]

```
bool process_crc = False
```

Definition at line 42 of file libGPS.py.

6.1.4.28 process_crc [2/2]

```
int process_crc = 2:
```

Definition at line 567 of file libGPS.py.

6.1.4.29 satellite_data

```
satellite_data = dict()
```

Definition at line 78 of file libGPS.py.

6.1.4.30 satellites_in_use

```
int satellites_in_use = 0
```

Definition at line 74 of file libGPS.py.

6.1.4.31 satellites_in_view

```
int satellites_in_view = 0
```

Definition at line 73 of file libGPS.py.

6.1.4.32 satellites_used

```
list satellites_used = []
```

Definition at line 75 of file libGPS.py.

6.1.4.33 sentence_active [1/2]

```
bool sentence_active = False
```

Definition at line 40 of file libGPS.py.

6.1.4.34 sentence_active [2/2]

```
str sentence_active = '*':
```

Definition at line 547 of file libGPS.py.

6.1.4.35 SENTENCE_LIMIT

```
int SENTENCE_LIMIT = 90  [static]
```

Definition at line 18 of file libGPS.py.

6.1.4.36 speed

```
list speed = [0.0, 0.0, 0.0]
```

Definition at line 67 of file libGPS.py.

6.1.4.37 supported_sentences [1/2]

```
int supported_sentences = 1
```

Definition at line 588 of file libGPS.py.

6.1.4.38 supported_sentences [2/2]

```
dict supported_sentences  [static]
```

Initial value:

```
=  {'GPRMC': gprmc, 'GLRMC': gprmc,
                          'GPGGA': gpgga, 'GLGGA': gpgga,
                          'GPVTG': gpvtg, 'GLVTG': gpvtg,
                          'GPGSA': gpgsa, 'GLGSA': gpgsa,
                          'GPGSV': gpgsv, 'GLGSV': gpgsv,
                          'GPGLL': gpgll, 'GLGLL': gpgll,
                          'GNGGA': gpgga, 'GNRMC': gprmc,
                          'GNVTG': gpvtg, 'GNGLL': gpgll,
                          'GNGSA': gpgsa,
                         }
```

Definition at line 759 of file libGPS.py.

6.1.4.39 timestamp

```
list timestamp = [0,0,0.0]
```

Definition at line 59 of file libGPS.py.

6.1.4.40 total_sv_sentences

```
int total_sv_sentences = 0
```

Definition at line 77 of file libGPS.py.

6.1.4.41 valid

```
bool valid = False
```

Definition at line 82 of file libGPS.py.

6.1.4.42 vdop

```
float vdop = 0.0
```

Definition at line 81 of file libGPS.py.

The documentation for this class was generated from the following file:

- Code/libGPS.py

Chapter 7

File Documentation

7.1 Code/libGPS.py File Reference

Data Structures

- class MicropyGPS

Namespaces

- namespace libGPS

7.2 libGPS.py

Go to the documentation of this file.

```
00001 from math import floor, modf
00002
00003 # Import utime or time for fix time handling
00004 try:
00005     # Assume running on MicroPython
00006     import utime
00007 except ImportError:
00008     # Otherwise default to time module for non-embedded implementations
00009     # Should still support millisecond resolution.
00010     import time
00011
00012
00013 class MicropyGPS(object):
00014     """GPS NMEA Sentence Parser. Creates object that stores all relevant GPS data and statistics.
00015     Parses sentences one character at a time using update(). """
00016
00017     # Max Number of Characters a valid sentence can be (based on GGA sentence)
00018     SENTENCE_LIMIT = 90
00019     __HEMISPHERES = ('N', 'S', 'E', 'W')
00020     __NO_FIX = 1
00021     __FIX_2D = 2
00022     __FIX_3D = 3
00023     __DIRECTIONS = ('N', 'NNE', 'NE', 'ENE', 'E', 'ESE', 'SE', 'SSE', 'S', 'SSW', 'SW', 'WSW', 'W',
00024                     'WNW', 'NW', 'NNW')
00025     __MONTHS = ('January', 'February', 'March', 'April', 'May',
00026                 'June', 'July', 'August', 'September', 'October',
00027                 'November', 'December')
00028
00029     def __init__(self, local_offset=0, location_formatting='dd'):
00030         """
00031         Setup GPS Object Status Flags, Internal Data Registers, etc
00032             local_offset (int): Timzone Difference to UTC
00033             location_formatting (str): Style For Presenting Longitude/Latitude:
00034                                        Decimal Degree Minute (ddm) - 40° 26.767 N
```

```
00035                                       Degrees Minutes Seconds (dms) - 40° 26 46 N
00036                                       Decimal Degrees (dd) - 40.446° N
00037         """
00038
00039         # Object Status Flags
00040         self.sentence_activesentence_active = False
00041         self.active_segment = 0
00042         self.process_crcprocess_crc = False
00043         self.gps_segments = []
00044         self.crc_xorcrc_xor = 0
00045         self.char_count = 0
00046         self.fix_time = 0
00047
00048         # Sentence Statistics
00049         self.crc_fails = 0
00050         self.clean_sentences = 0
00051         self.parsed_sentences = 0
00052
00053         # Logging Related
00054         self.log_handle = None
00055         self.log_enlog_en = False
00056
00057         # Data From Sentences
00058         # Time
00059         self.timestamp = [0,0,0.0]
00060         self.datedate = [0, 0, 0]
00061         self.local_offset = local_offset
00062
00063         # Position/Motion
00064         self._latitude = [0, 0.0, 'N']
00065         self._longitude = [0, 0.0, 'W']
00066         self.coord_format = location_formatting
00067         self.speed = [0.0, 0.0, 0.0]
00068         self.course = 0.0
00069         self.altitude = 0.0
00070         self.geoid_height = 0.0
00071
00072         # GPS Info
00073         self.satellites_in_view = 0
00074         self.satellites_in_use = 0
00075         self.satellites_used = []
00076         self.last_sv_sentence = 0
00077         self.total_sv_sentences = 0
00078         self.satellite_data = dict()
00079         self.hdop = 0.0
00080         self.pdop = 0.0
00081         self.vdop = 0.0
00082         self.valid = False
00083         self.fix_stat = 0
00084         self.fix_type = 1
00085
00086
00088     @property
00089     def latitude(self):
00090         """Format Latitude Data Correctly"""
00091         if self.coord_format == 'dd':
00092             decimal_degrees = self._latitude[0] + (self._latitude[1] / 60)
00093             return [decimal_degrees, self._latitude[2]]
00094         elif self.coord_format == 'dms':
00095             minute_parts = modf(self._latitude[1])
00096             seconds = round(minute_parts[0] * 60)
00097             return [self._latitude[0], int(minute_parts[1]), seconds, self._latitude[2]]
00098         else:
00099             return self._latitude
00100
00101     @property
00102     def longitude(self):
00103         """Format Longitude Data Correctly"""
00104         if self.coord_format == 'dd':
00105             decimal_degrees = self._longitude[0] + (self._longitude[1] / 60)
00106             return [decimal_degrees, self._longitude[2]]
00107         elif self.coord_format == 'dms':
00108             minute_parts = modf(self._longitude[1])
00109             seconds = round(minute_parts[0] * 60)
00110             return [self._longitude[0], int(minute_parts[1]), seconds, self._longitude[2]]
00111         else:
00112             return self._longitude
00113
00114
00116     def start_logging(self, target_file, mode="append"):
00117         """Create GPS data log object"""
00118         # Set Write Mode Overwrite or Append
00119         mode_code = 'w' if mode == 'new' else 'a'
00120
00121         try:
00122             self.log_handle = open(target_file, mode_code)
00123         except AttributeError:
```

```
00124             print("Invalid FileName")
00125             return False
00126 
00127         self.log_enlog_en = True
00128         return True
00129 
00130     def stop_logging(self):
00131         """Closes the log file handler and disables further logging"""
00132         try:
00133             self.log_handle.close()
00134         except AttributeError:
00135             print("Invalid Handle")
00136             return False
00137 
00138         self.log_enlog_en = False
00139         return True
00140 
00141     def write_log(self, log_string):
00142         """Attempts to write the last valid NMEA sentence character to the active file handler"""
00143         try:
00144             self.log_handle.write(log_string)
00145         except TypeError:
00146             return False
00147         return True
00148 
00149     ########################################
00150     # Sentence Parsers
00151     def gprmc(self):
00152         """Parse Recommended Minimum Specific GPS/Transit data (RMC)Sentence.
00153         Updates UTC timestamp, latitude, longitude, Course, Speed, Date, and fix status"""
00154 
00155         # UTC Timestamp
00156         try:
00157             utc_string = self.gps_segments[1]
00158 
00159             if utc_string:  # Possible timestamp found
00160                 hours = (int(utc_string[0:2]) + self.local_offset) % 24
00161                 minutes = int(utc_string[2:4])
00162                 seconds = float(utc_string[4:])
00163                 self.timestamp = f"{hours:02}:{minutes:02}:{seconds:05.2f}"
00164             else:  # No Time stamp yet
00165                 self.timestamp = [0,0,0.0]
00166 
00167         except ValueError:  # Bad Timestamp value present
00168             return False
00169 
00170         # Date stamp
00171         try:
00172             date_string = self.gps_segments[9]
00173 
00174             # Date string printer function assumes to be year >=2000,
00175             # date_string() must be supplied with the correct century argument to display correctly
00176             if date_string:  # Possible date stamp found
00177                 day = int(date_string[0:2])
00178                 month = int(date_string[2:4])
00179                 year = int(date_string[4:6])
00180                 self.date = (day, month, year)
00181             else:  # No Date stamp yet
00182                 self.date = (0, 0, 0)
00183 
00184         except ValueError:  # Bad Date stamp value present
00185             return False
00186 
00187         # Check Receiver Data Valid Flag
00188         if self.gps_segments[2] == 'A':  # Data from Receiver is Valid/Has Fix
00189 
00190             # Longitude / Latitude
00191             try:
00192                 # Latitude
00193                 l_string = self.gps_segments[3]
00194                 lat_degs = int(l_string[0:2])
00195                 lat_mins = float(l_string[2:])
00196                 lat_hemi = self.gps_segments[4]
00197 
00198                 # Longitude
00199                 l_string = self.gps_segments[5]
00200                 lon_degs = int(l_string[0:3])
00201                 lon_mins = float(l_string[3:])
00202                 lon_hemi = self.gps_segments[6]
00203             except ValueError:
00204                 return False
00205 
00206             if lat_hemi not in self.__HEMISPHERES:
00207                 return False
00208 
00209             if lon_hemi not in self.__HEMISPHERES:
00210                 return False
```

```
00211
00212             # Speed
00213             try:
00214                 spd_knt = float(self.gps_segments[7])
00215             except ValueError:
00216                 return False
00217
00218             # Course
00219             try:
00220                 if self.gps_segments[8]:
00221                     course = float(self.gps_segments[8])
00222                 else:
00223                     course = 0.0
00224             except ValueError:
00225                 return False
00226
00227             # Update Object Data
00228             self._latitude = [lat_degs, lat_mins, lat_hemi]
00229             self._longitude = [lon_degs, lon_mins, lon_hemi]
00230             # Include mph and hm/h
00231             self.speed = [spd_knt, spd_knt * 1.151, spd_knt * 1.852]
00232             self.course = course
00233             self.valid = True
00234
00235             # Update Last Fix Time
00236             self.new_fix_time()
00237
00238         else:  # Clear Position Data if Sentence is 'Invalid'
00239             self._latitude = [0, 0.0, 'N']
00240             self._longitude = [0, 0.0, 'W']
00241             self.speed = [0.0, 0.0, 0.0]
00242             self.course = 0.0
00243             self.valid = False
00244
00245         return True
00246
00247     def gpgll(self):
00248         """Parse Geographic Latitude and Longitude (GLL)Sentence. Updates UTC timestamp, latitude,
      longitude, and fix status"""
00249
00250         # UTC Timestamp
00251         try:
00252             utc_string = self.gps_segments[5]
00253
00254             if utc_string:  # Possible timestamp found
00255                 hours = (int(utc_string[0:2]) + self.local_offset) % 24
00256                 minutes = int(utc_string[2:4])
00257                 seconds = float(utc_string[4:])
00258                 self.timestamp = f"{hours:02}:{minutes:02}:{seconds:05.2f}"
00259             else:  # No Time stamp yet
00260                 self.timestamp = [0,0,0.0]
00261
00262         except ValueError:  # Bad Timestamp value present
00263             return False
00264
00265         # Check Receiver Data Valid Flag
00266         if self.gps_segments[6] == 'A':  # Data from Receiver is Valid/Has Fix
00267
00268             # Longitude / Latitude
00269             try:
00270                 # Latitude
00271                 l_string = self.gps_segments[1]
00272                 lat_degs = int(l_string[0:2])
00273                 lat_mins = float(l_string[2:])
00274                 lat_hemi = self.gps_segments[2]
00275
00276                 # Longitude
00277                 l_string = self.gps_segments[3]
00278                 lon_degs = int(l_string[0:3])
00279                 lon_mins = float(l_string[3:])
00280                 lon_hemi = self.gps_segments[4]
00281             except ValueError:
00282                 return False
00283
00284             if lat_hemi not in self.__HEMISPHERES:
00285                 return False
00286
00287             if lon_hemi not in self.__HEMISPHERES:
00288                 return False
00289
00290             # Update Object Data
00291             self._latitude = [lat_degs, lat_mins, lat_hemi]
00292             self._longitude = [lon_degs, lon_mins, lon_hemi]
00293             self.valid = True
00294
00295             # Update Last Fix Time
00296             self.new_fix_time()
```

```
00297
00298         else:  # Clear Position Data if Sentence is 'Invalid'
00299             self._latitude = [0, 0.0, 'N']
00300             self._longitude = [0, 0.0, 'W']
00301             self.valid = False
00302
00303         return True
00304
00305     def gpvtg(self):
00306         """Parse Track Made Good and Ground Speed (VTG) Sentence. Updates speed and course"""
00307         try:
00308             course = float(self.gps_segments[1]) if self.gps_segments[1] else 0.0
00309             spd_knt = float(self.gps_segments[5]) if self.gps_segments[5] else 0.0
00310         except ValueError:
00311             return False
00312
00313         # Include mph and km/h
00314         self.speed = (spd_knt, spd_knt * 1.151, spd_knt * 1.852)
00315         self.course = course
00316         return True
00317
00318     def gpgga(self):
00319         """Parse Global Positioning System Fix Data (GGA) Sentence. Updates UTC timestamp, latitude,
    longitude,
00320         fix status, satellites in use, Horizontal Dilution of Precision (HDOP), altitude, geoid height
    and fix status"""
00321
00322         try:
00323             # UTC Timestamp
00324             utc_string = self.gps_segments[1]
00325
00326             # Skip timestamp if receiver doesn't have on yet
00327             if utc_string:
00328                 hours = (int(utc_string[0:2]) + self.local_offset) % 24
00329                 minutes = int(utc_string[2:4])
00330                 seconds = float(utc_string[4:])
00331             else:
00332                 hours = 0
00333                 minutes = 0
00334                 seconds = 0.0
00335
00336             # Number of Satellites in Use
00337             satellites_in_use = int(self.gps_segments[7])
00338
00339             # Get Fix Status
00340             fix_stat = int(self.gps_segments[6])
00341
00342         except (ValueError, IndexError):
00343             return False
00344
00345         try:
00346             # Horizontal Dilution of Precision
00347             hdop = float(self.gps_segments[8])
00348         except (ValueError, IndexError):
00349             hdop = 0.0
00350
00351         # Process Location and Speed Data if Fix is GOOD
00352         if fix_stat:
00353
00354             # Longitude / Latitude
00355             try:
00356                 # Latitude
00357                 l_string = self.gps_segments[2]
00358                 lat_degs = int(l_string[0:2])
00359                 lat_mins = float(l_string[2:])
00360                 lat_hemi = self.gps_segments[3]
00361
00362                 # Longitude
00363                 l_string = self.gps_segments[4]
00364                 lon_degs = int(l_string[0:3])
00365                 lon_mins = float(l_string[3:])
00366                 lon_hemi = self.gps_segments[5]
00367             except ValueError:
00368                 return False
00369
00370             if lat_hemi not in self.__HEMISPHERES:
00371                 return False
00372
00373             if lon_hemi not in self.__HEMISPHERES:
00374                 return False
00375
00376             # Altitude / Height Above Geoid
00377             try:
00378                 altitude = float(self.gps_segments[9])
00379                 geoid_height = float(self.gps_segments[11])
00380             except ValueError:
00381                 altitude = 0
```

```
00382                 geoid_height = 0
00383
00384             # Update Object Data
00385             self._latitude = [lat_degs, lat_mins, lat_hemi]
00386             self._longitude = [lon_degs, lon_mins, lon_hemi]
00387             self.altitude = altitude
00388             self.geoid_height = geoid_height
00389
00390         # Update Object Data
00391         self.timestamp = f"{hours:02}:{minutes:02}:{seconds:05.2f}"
00392         self.satellites_in_use = satellites_in_use
00393         self.hdop = hdop
00394         self.fix_stat = fix_stat
00395
00396         # If Fix is GOOD, update fix timestamp
00397         if fix_stat:
00398             self.new_fix_time()
00399
00400         return True
00401
00402     def gpgsa(self):
00403         """Parse GNSS DOP and Active Satellites (GSA) sentence. Updates GPS fix type, list of
      satellites used in
00404         fix calculation, Position Dilution of Precision (PDOP), Horizontal Dilution of Precision
      (HDOP), Vertical
00405         Dilution of Precision (VDOP), and fix status"""
00406
00407         # Fix Type (None,2D or 3D)
00408         try:
00409             fix_type = int(self.gps_segments[2])
00410         except ValueError:
00411             return False
00412
00413         # Read All (up to 12) Available PRN Satellite Numbers
00414         sats_used = []
00415         for sats in range(12):
00416             sat_number_str = self.gps_segments[3 + sats]
00417             if sat_number_str:
00418                 try:
00419                     sat_number = int(sat_number_str)
00420                     sats_used.append(sat_number)
00421                 except ValueError:
00422                     return False
00423             else:
00424                 break
00425
00426         # PDOP,HDOP,VDOP
00427         try:
00428             pdop = float(self.gps_segments[15])
00429             hdop = float(self.gps_segments[16])
00430             vdop = float(self.gps_segments[17])
00431         except ValueError:
00432             return False
00433
00434         # Update Object Data
00435         self.fix_type = fix_type
00436
00437         # If Fix is GOOD, update fix timestamp
00438         if fix_type > self.__NO_FIX:
00439             self.new_fix_time()
00440
00441         self.satellites_used = sats_used
00442         self.hdop = hdop
00443         self.vdop = vdop
00444         self.pdop = pdop
00445
00446         return True
00447
00448     def gpgsv(self):
00449         """Parse Satellites in View (GSV) sentence. Updates number of SV Sentences,the number of the
      last SV sentence
00450         parsed, and data on each satellite present in the sentence"""
00451         try:
00452             num_sv_sentences = int(self.gps_segments[1])
00453             current_sv_sentence = int(self.gps_segments[2])
00454             sats_in_view = int(self.gps_segments[3])
00455         except ValueError:
00456             return False
00457
00458         # Create a blank dict to store all the satellite data from this sentence in:
00459         # satellite PRN is key, tuple containing telemetry is value
00460         satellite_dict = dict()
00461
00462         # Calculate  Number of Satelites to pull data for and thus how many segment positions to read
00463         if num_sv_sentences == current_sv_sentence:
00464             # Last sentence may have 1-4 satellites; 5 - 20 positions
00465             sat_segment_limit = (sats_in_view - ((num_sv_sentences - 1) * 4)) * 5
```

```
00466         else:
00467             sat_segment_limit = 20  # Non-last sentences have 4 satellites and thus read up to
      position 20
00468 
00469         # Try to recover data for up to 4 satellites in sentence
00470         for sats in range(4, sat_segment_limit, 4):
00471 
00472             # If a PRN is present, grab satellite data
00473             if self.gps_segments[sats]:
00474                 try:
00475                     sat_id = int(self.gps_segments[sats])
00476                 except (ValueError,IndexError):
00477                     return False
00478 
00479                 try:  # elevation can be null (no value) when not tracking
00480                     elevation = int(self.gps_segments[sats+1])
00481                 except (ValueError,IndexError):
00482                     elevation = None
00483 
00484                 try:  # azimuth can be null (no value) when not tracking
00485                     azimuth = int(self.gps_segments[sats+2])
00486                 except (ValueError,IndexError):
00487                     azimuth = None
00488 
00489                 try:  # SNR can be null (no value) when not tracking
00490                     snr = int(self.gps_segments[sats+3])
00491                 except (ValueError,IndexError):
00492                     snr = None
00493             # If no PRN is found, then the sentence has no more satellites to read
00494             else:
00495                 break
00496 
00497             # Add Satellite Data to Sentence Dict
00498             satellite_dict[sat_id] = (elevation, azimuth, snr)
00499 
00500         # Update Object Data
00501         self.total_sv_sentences = num_sv_sentences
00502         self.last_sv_sentence = current_sv_sentence
00503         self.satellites_in_view = sats_in_view
00504 
00505         # For a new set of sentences, we either clear out the existing sat data or
00506         # update it as additional SV sentences are parsed
00507         if current_sv_sentence == 1:
00508             self.satellite_data = satellite_dict
00509         else:
00510             self.satellite_data.update(satellite_dict)
00511 
00512         return True
00513 
00514     ##########################################
00515     # Data Stream Handler Functions
00516     def new_sentence(self):
00517         """Adjust Object Flags in Preparation for a New Sentence"""
00518         self.gps_segments = ['']
00519         self.active_segment = 0
00520         self.crc_xor = 0
00521         self.sentence_active = True
00522         self.process_crc = True
00523         self.char_count = 0
00524 
00525     def update(self, new_char):
00526         """Process a new input char and updates GPS object if necessary based on special characters
      ('$', ',', '*')
00527         Function builds a list of received string that are validate by CRC prior to parsing by the
      appropriate
00528         sentence function. Returns sentence type on successful parse, None otherwise"""
00529 
00530         valid_sentence = False
00531 
00532         # Validate new_char is a printable char
00533         ascii_char = ord(new_char)
00534 
00535         if 10 <= ascii_char <= 126:
00536             self.char_count += 1
00537 
00538             # Write Character to log file if enabled
00539             if self.log_en:
00540                 self.write_log(new_char)
00541 
00542             # Check if a new string is starting ($)
00543             if new_char == '$':
00544                 self.new_sentence()
00545                 return None
00546 
00547             elif self.sentence_active:
00548 
00549                 # Check if sentence is ending (*)
```

```
00550                 if new_char == '*':
00551                     self.process_crc = False
00552                     self.active_segment += 1
00553                     self.gps_segments.append('')
00554                     return None
00555 
00556                 # Check if a section is ended (,), Create a new substring to feed
00557                 # characters to
00558                 elif new_char == ',':
00559                     self.active_segment += 1
00560                     self.gps_segments.append('')
00561 
00562                 # Store All Other printable character and check CRC when ready
00563                 else:
00564                     self.gps_segments[self.active_segment] += new_char
00565 
00566                     # When CRC input is disabled, sentence is nearly complete
00567                     if not self.process_crc:
00568 
00569                         if len(self.gps_segments[self.active_segment]) == 2:
00570                             try:
00571                                 final_crc = int(self.gps_segments[self.active_segment], 16)
00572                                 if self.crc_xor == final_crc:
00573                                     valid_sentence = True
00574                                 else:
00575                                     self.crc_fails += 1
00576                             except ValueError:
00577                                 pass  # CRC Value was deformed and could not have been correct
00578 
00579                 # Update CRC
00580                 if self.process_crc:
00581                     self.crc_xor ^= ascii_char
00582 
00583                 # If a Valid Sentence Was received and it's a supported sentence, then parse it!!
00584                 if valid_sentence:
00585                     self.clean_sentences += 1  # Increment clean sentences received
00586                     self.sentence_active = False  # Clear Active Processing Flag
00587 
00588                     if self.gps_segments[0] in self.supported_sentences:
00589 
00590                         # parse the Sentence Based on the message type, return True if parse is clean
00591                         if self.supported_sentences[self.gps_segments[0]](self):
00592 
00593                             # Let host know that the GPS object was updated by returning parsed
      sentence type
00594                             self.parsed_sentences += 1
00595                             return self.gps_segments[0]
00596 
00597                 # Check that the sentence buffer isn't filling up with Garage waiting for the sentence
      to complete
00598                 if self.char_count > self.SENTENCE_LIMIT:
00599                     self.sentence_active = False
00600 
00601         # Tell Host no new sentence was parsed
00602         return None
00603 
00604     def new_fix_time(self):
00605         """Updates a high resolution counter with current time when fix is updated. Currently only
      triggered from
00606         GGA, GSA and RMC sentences"""
00607         try:
00608             self.fix_time = utime.ticks_ms()
00609         except NameError:
00610             self.fix_time = time.time()
00611 
00612     #########################################
00613     # User Helper Functions working with the GPS object data easier
00614     def satellite_data_updated(self):
00615         """Checks if the all the GSV sentences in a group have been read, making satellite data
      complete:return: boolean"""
00616         if self.total_sv_sentences > 0 and self.total_sv_sentences == self.last_sv_sentence:
00617             return True
00618         else:
00619             return False
00620 
00621     def unset_satellite_data_updated(self):
00622         """Mark GSV sentences as read indicating the data has been used and future updates are
      fresh"""
00623         self.last_sv_sentence = 0
00624 
00625     def satellites_visible(self):
00626         """Returns a list of of the satellite PRNs currently visible to the receiver:return: list"""
00627         return list(self.satellite_data.keys())
00628 
00629     def time_since_fix(self):
00630         """Returns number of millisecond since the last sentence with a valid fix was parsed. Returns
      0 if no fix has been found"""
```

```
00631         # Test if a Fix has been found
00632         if self.fix_time == 0:
00633             return -1
00634 
00635         # Try calculating fix time using utime; if not running MicroPython
00636         # time.time() returns a floating point value in secs
00637         try:
00638             current = utime.ticks_diff(utime.ticks_ms(), self.fix_time)
00639         except NameError:
00640             current = (time.time() - self.fix_time) * 1000  # ms
00641 
00642         return current
00643 
00644     def compass_direction(self):
00645         """Determine a cardinal or inter-cardinal direction based on current course.:return: string"""
00646         # Calculate the offset for a rotated compass
00647         if self.course >= 348.75:
00648             offset_course = 360 - self.course
00649         else:
00650             offset_course = self.course + 11.25
00651 
00652         # Each compass point is separated by 22.5 degrees, divide to find lookup value
00653         dir_index = floor(offset_course / 22.5)
00654 
00655         final_dir = self.__DIRECTIONS[dir_index]
00656 
00657         return final_dir
00658 
00659     def latitude_string(self):
00660         """Create a readable string of the current latitude data:return: string"""
00661         if self.coord_format == 'dd':
00662             formatted_latitude = self.latitude
00663             lat_string = str(formatted_latitude[0]) + 'g ' + str(self._latitude[2])
00664         elif self.coord_format == 'dms':
00665             formatted_latitude = self.latitude
00666             lat_string = str(formatted_latitude[0]) + 'g ' + str(formatted_latitude[1]) + "' " +
      str(formatted_latitude[2]) + '" ' + str(formatted_latitude[3])
00667         else:
00668             lat_string = str(self._latitude[0]) + 'g ' + str(self._latitude[1]) + "' " +
      str(self._latitude[2])
00669         return lat_string
00670 
00671     def longitude_string(self):
00672         """Create a readable string of the current longitude data: return: string"""
00673         if self.coord_format == 'dd':
00674             formatted_longitude = self.longitude
00675             lon_string = str(formatted_longitude[0]) + 'g ' + str(self._longitude[2])
00676         elif self.coord_format == 'dms':
00677             formatted_longitude = self.longitude
00678             lon_string = str(formatted_longitude[0]) + 'g ' + str(formatted_longitude[1]) + "' " +
      str(formatted_longitude[2]) + '" ' + str(formatted_longitude[3])
00679         else:
00680             lon_string = str(self._longitude[0]) + 'g ' + str(self._longitude[1]) + "' " +
      str(self._longitude[2])
00681         return lon_string
00682 
00683     def speed_string(self, unit='kph'):
00684         """
00685         Creates a readable string of the current speed data in one of three units
00686         :param unit: string of 'kph' or 'mph
00687         :return:
00688         """
00689         if unit == 'mph':
00690             speed_string = f"{self.speed[1]:.2f} mph"
00691 
00692         else:
00693             speed_string = f"{self.speed[2]:.2f} km/h"
00694 
00695         return speed_string
00696 
00697     def date_string(self, formatting='s_dmy', century='20'):
00698         """
00699         Creates a readable string of the current date.
00700         Can select between long format: Januray 1st, 2014
00701         or two short formats:
00702         11/01/2014 (MM/DD/YYYY)
00703         01/11/2014 (DD/MM/YYYY)
00704         :param formatting: string 's_mdy', 's_dmy', or 'long'
00705         :param century: int delineating the century the GPS data is from (19 for 19XX, 20 for 20XX)
00706         :return: date_string  string with long or short format date
00707         """
00708 
00709         # Long Format Januray 1st, 2014
00710         if formatting == 'long':
00711             # Retrieve Month string from private set
00712             month = self.__MONTHS[self.date[1] - 1]
00713 
```

```
00714             # Determine Date Suffix
00715             if self.date[0] in (1, 21, 31):
00716                 suffix = 'st'
00717             elif self.date[0] in (2, 22):
00718                 suffix = 'nd'
00719             elif self.date[0] == (3, 23):
00720                 suffix = 'rd'
00721             else:
00722                 suffix = 'th'
00723 
00724             day = str(self.date[0]) + suffix  # Create Day String
00725 
00726             year = century + str(self.date[2])  # Create Year String
00727 
00728             date_string = month + ' ' + day + ', ' + year  # Put it all together
00729 
00730         else:
00731             # Add leading zeros to day string if necessary
00732             if self.date[0] < 10:
00733                 day = '0' + str(self.date[0])
00734             else:
00735                 day = str(self.date[0])
00736 
00737             # Add leading zeros to month string if necessary
00738             if self.date[1] < 10:
00739                 month = '0' + str(self.date[1])
00740             else:
00741                 month = str(self.date[1])
00742 
00743             # Add leading zeros to year string if necessary
00744             if self.date[2] < 10:
00745                 year = '0' + str(self.date[2])
00746             else:
00747                 year = str(self.date[2])
00748 
00749             # Build final string based on desired formatting
00750             if formatting == 's_dmy':
00751                 date_string = day + '/' + month + '/' + year
00752 
00753             else:  # Default date format
00754                 date_string = month + '/' + day + '/' + year
00755 
00756         return date_string
00757 
00758     # All the currently supported NMEA sentences
00759     supported_sentences = {'GPRMC': gprmc, 'GLRMC': gprmc,
00760                            'GPGGA': gpgga, 'GLGGA': gpgga,
00761                            'GPVTG': gpvtg, 'GLVTG': gpvtg,
00762                            'GPGSA': gpgsa, 'GLGSA': gpgsa,
00763                            'GPGSV': gpgsv, 'GLGSV': gpgsv,
00764                            'GPGLL': gpgll, 'GLGLL': gpgll,
00765                            'GNGGA': gpgga, 'GNRMC': gprmc,
00766                            'GNVTG': gpvtg, 'GNGLL': gpgll,
00767                            'GNGSA': gpgsa,
00768                           }
00769 
00770 if __name__ == "__main__":
00771     pass
```

7.3 Code/Main_v2.0.py File Reference

Namespaces

- namespace Main_v2

Functions

- initialize_display ()
- update_display (oled, gps_data)
- parse_gps_data (my_gps)
- main ()

Variables

- int I2C_SCL_PIN = 22
- int I2C_SDA_PIN = 21
- int I2C_FREQ = 100_000
- int UART_TX_PIN = 17
- int UART_RX_PIN = 16
- int UART_BAUDRATE = 9600
- int OLED_CONTRAST = 80
- int DISPLAY_REFRESH_INTERVAL = 1

7.4 Main_v2.0.py

Go to the documentation of this file.

```
00001 """
00002 ================================================================================
00003 Project Name: GPS Data Display on OLED
00004 File Name: Main_v2.0.py
00005 Description: This script reads GPS data from a UART-connected GPS module, parses
00006              the data using MicropyGPS, and displays it on an SH1106 OLED screen.
00007              The displayed information includes time, date, latitude, longitude,
00008              altitude, number of satellites, and horizontal dilution of precision (HDOP).
00009
00010 Authors:
00011   - Guilherme Brito
00012   - Henrique Silva
00013   - João Santos
00014
00015 Version: 2.0
00016 Date Created: 14-11-2024
00017 Last Modified: 21-11-2024
00018
00019 Parameters:
00020   - I2C Pins: SCL = GP22, SDA = GP21
00021   - UART Pins: TX = GP17, RX = GP16
00022   - OLED Contrast: 80%
00023   - GPS Baud Rate: 9600
00024   - Display Refresh Interval: 1 second
00025
00026 Usage:
00027   - Ensure the SH1106 OLED display and GPS module are connected as per the pin
00028     configuration.
00029   - Upload the script to a microcontroller and monitor the OLED for GPS data.
00030   - Exit the program with Ctrl+C.
00031
00032 Dependencies:
00033   - machine (for I2C and UART)
00034   - sh1106 (OLED display driver)
00035   - libGPS (Adaptation of the MicropyGPS for GPS parsing)
00036
00037 Notes:
00038   - This code is optimize for ESP32
00039   - See documentation for connections between the three
00040
00041 ================================================================================
00042 """
00043
00044 import machine
00045 from machine import I2C, Pin
00046 from sh1106 import SH1106_I2C
00047 from time import sleep
00048 from libGPS import MicropyGPS
00049
00050 # Constants
00051 I2C_SCL_PIN = 22  # Pin for I2C clock line
00052 I2C_SDA_PIN = 21  # Pin for I2C data line
00053 I2C_FREQ = 100_000  # Frequency for I2C communication in Hz
00054 UART_TX_PIN = 17  # Pin for UART TX
00055 UART_RX_PIN = 16  # Pin for UART RX
00056 UART_BAUDRATE = 9600  # Baud rate for UART communication
00057 OLED_CONTRAST = 80  # Contrast setting for the OLED display (0-255)
00058 DISPLAY_REFRESH_INTERVAL = 1  # Interval for refreshing the display in seconds
00059
00060 # Function: initialize_display
00061 # Description: Initializes the OLED display via I2C.
00062 # Returns: The initialized OLED object.
```

```
00063 # Raises: Exception if the I2C or OLED initialization fails.
00064 def initialize_display():
00065     """
00066     Set up the I2C bus and OLED display.
00067     """
00068     try:
00069         # Initialize I2C bus
00070         i2c = I2C(0, scl=Pin(I2C_SCL_PIN), sda=Pin(I2C_SDA_PIN), freq=I2C_FREQ)
00071         # Initialize OLED display
00072         oled = SH1106_I2C(i2c)
00073         oled.contrast(OLED_CONTRAST)
00074         return oled
00075     except Exception as e:
00076         print(f"Error initializing OLED: {e}")
00077         raise
00078
00079 # Function: update_display
00080 # Description: Updates the OLED display with the given GPS data.
00081 # Parameters:
00082 #   oled (SH1106_I2C): The OLED display object.
00083 #   gps_data (dict): Dictionary containing GPS data (time, date, lat, long, etc.).
00084 def update_display(oled, gps_data):
00085     """
00086     Refreshes the OLED display with new GPS data.
00087     """
00088     oled.fill(0)  # Clear the screen
00089
00090     # Display labels
00091     oled.text("Date:", 0, 0) #(text, x position, y position)
00092     oled.text("Time:", 0, 9)
00093     oled.text("Lat:", 0, 18)
00094     oled.text("Long:", 0, 27)
00095     oled.text("Altitude:", 0, 36)
00096     oled.text("Speed:", 0, 45)
00097     oled.text("N Satellites:", 0, 54)
00098
00099     # Display GPS data
00100     oled.text(f"{gps_data['date']}", 38, 0)  # Truncate time to HH:MM:SS
00101     oled.text(f"{gps_data['time']}", 38, 9)
00102     oled.text(f"{gps_data['latitude'][:12]}", 30, 18)
00103     oled.text(f"{gps_data['longitude'][:12]}", 38, 27)
00104     oled.text(f"{gps_data['altitude']} m", 70, 36)
00105     oled.text(f"{gps_data['speed']}", 47, 45)
00106     oled.text(f"{gps_data['satellites']}", 103, 54)
00107     oled.show()  # Update the OLED screen
00108
00109
00110 # Function: parse_gps_data
00111 # Description: Parses raw GPS data from the MicropyGPS object into a dictionary.
00112 # Parameters:
00113 #   my_gps (MicropyGPS): The MicropyGPS object handling GPS data parsing.
00114 # Returns: Dictionary containing parsed GPS data.
00115 def parse_gps_data(my_gps):
00116     """
00117     Extracts GPS data from the MicropyGPS object and formats it.
00118     """
00119     return {
00120         'date': my_gps.date_string('s_dmy'),  # Date in DD/MM/YYYY format
00121         'time': my_gps.timestamp,  # Format time as HH:MM:SS
00122         'latitude': my_gps.latitude_string(),  # Latitude in degrees/minutes/seconds
00123         'longitude': my_gps.longitude_string(),  # Longitude in degrees/minutes/seconds
00124         'altitude': str(my_gps.altitude),  # Altitude in meters
00125         'speed': my_gps.speed_string(),  # Longitude in degrees/minutes/seconds
00126         'satellites': str(my_gps.satellites_in_use),  # Number of satellites in use
00127         'pdop': str(my_gps.pdop),  # Horizontal Dilution of Precision
00128         'hdop': str(my_gps.hdop),  # Horizontal Dilution of Precision
00129         'vdop': str(my_gps.vdop),  # Horizontal Dilution of Precision
00130     }
00131
00132 # Function: main
00133 # Description: Main function to initialize components and handle GPS data processing and display
      updates.
00134 def main():
00135     """
00136     The main program loop. Initializes the OLED display and GPS module, then continuously
00137     updates the display with parsed GPS data.
00138     """
00139     # Initialize the OLED display
00140     oled = initialize_display()
00141
00142     # Instantiate the GPS parser
00143     my_gps = MicropyGPS()
00144
00145     # Initialize UART for GPS communication
00146     gps_serial = machine.UART(2, baudrate=UART_BAUDRATE, tx=UART_TX_PIN, rx=UART_RX_PIN)
00147
00148     print("System initialized. Press Ctrl+C to exit.")
```

```
00149     try:
00150         while True:
00151             gps_data_available = False
00152             # Read GPS data from the UART interface
00153             while gps_serial.any():
00154                 data = gps_serial.read()
00155                 for byte in data:
00156                     # Update GPS parser with each byte
00157                     if my_gps.update(chr(byte)) is not None:
00158                         gps_data_available = True
00159 
00160             # If new GPS data is available, process and display it
00161             if gps_data_available:
00162                 gps_data = parse_gps_data(my_gps)
00163                 update_display(oled, gps_data)
00164 
00165                 # Print GPS data line by line for readability
00166                 print("\n--- GPS Data ---")
00167                 for key, value in gps_data.items():
00168                     # Print GPS data line by line for readability
00169                     print("\n--- GPS Data ---")
00170                     print(f"Date: {gps_data['date']}")
00171                     print(f"Time: {gps_data['time']}")
00172                     print(f"Latitude: {gps_data['latitude']}")
00173                     print(f"Longitude: {gps_data['longitude']}")
00174                     print(f"Altitude: {gps_data['altitude']} meters")
00175                     print(f"Speed: {gps_data['speed']}")
00176                     print(f"Satellites: {gps_data['satellites']}")
00177                     print(f"Position Dilution of Precision: {gps_data['pdop']}")
00178                     print(f"Horizontal Dilution of Precision: {gps_data['hdop']}")
00179                     print(f"Vertical Dilution of Precision: {gps_data['vdop']}")
00180                 print("------------------")
00181 
00182             sleep(DISPLAY_REFRESH_INTERVAL)  # Delay to control refresh rate
00183     except KeyboardInterrupt:
00184         print("Program stopped. Exiting...")
00185         oled.poweroff()  # Turn off the OLED display
00186 
00187 # Entry point for the script
00188 if __name__ == "__main__":
00189     main()
```

7.5 Code/test.py File Reference

Namespaces

- namespace test

7.6 test.py

Go to the documentation of this file.

```
00001
```