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.

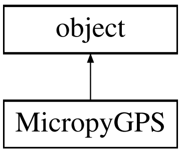
5.3 test Namespace Reference

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.

```
from math import floor, modf

# Import utime or time for fix time handling
try:
    # Assume running on MicroPython
    import utime
except ImportError:
    # Otherwise default to time module for non-embedded implementations
    # Should still support millisecond resolution.
    import time


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

    # Max Number of Characters a valid sentence can be (based on GGA sentence)
    SENTENCE_LIMIT = 90
    __HEMISPHERES = (’N’, ’S’, ’E’, ’W’)
    __NO_FIX = 1
    __FIX_2D = 2
    __FIX_3D = 3
    __DIRECTIONS = (’N’, ’NNE’, ’NE’, ’ENE’, ’E’, ’ESE’, ’SE’, ’SSE’, ’S’, ’SSW’, ’SW’, ’WSW’, ’W’,
                    ’WNW’, ’NW’, ’NNW’)
    __MONTHS = (’January’, ’February’, ’March’, ’April’, ’May’,
                ’June’, ’July’, ’August’, ’September’, ’October’,
                ’November’, ’December’)

    def __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
        """

        # Object Status Flags
        self.sentence_activesentence_active = False
        self.active_segment = 0
        self.process_crcprocess_crc = False
        self.gps_segments = []
        self.crc_xorcrc_xor = 0
        self.char_count = 0
        self.fix_time = 0

        # Sentence Statistics
        self.crc_fails = 0
        self.clean_sentences = 0
        self.parsed_sentences = 0

        # Logging Related
        self.log_handle = None
        self.log_enlog_en = False

        # Data From Sentences
        # Time
        self.timestamp = [0,0,0.0]
        self.datedate = [0, 0, 0]
        self.local_offset = local_offset

        # Position/Motion
        self._latitude = [0, 0.0, ’N’]
        self._longitude = [0, 0.0, ’W’]
        self.coord_format = location_formatting
        self.speed = [0.0, 0.0, 0.0]
        self.course = 0.0
        self.altitude = 0.0
        self.geoid_height = 0.0

        # GPS Info
        self.satellites_in_view = 0
        self.satellites_in_use = 0
        self.satellites_used = []
        self.last_sv_sentence = 0
        self.total_sv_sentences = 0
        self.satellite_data = dict()
        self.hdop = 0.0
        self.pdop = 0.0
        self.vdop = 0.0
        self.valid = False
        self.fix_stat = 0
        self.fix_type = 1


    @property
    def latitude(self):
        """Format Latitude Data Correctly"""
        if self.coord_format == ’dd’:
            decimal_degrees = self._latitude[0] + (self._latitude[1] / 60)
            return [decimal_degrees, self._latitude[2]]
        elif self.coord_format == ’dms’:
            minute_parts = modf(self._latitude[1])
            seconds = round(minute_parts[0] * 60)
            return [self._latitude[0], int(minute_parts[1]), seconds, self._latitude[2]]
        else:
            return self._latitude

    @property
    def longitude(self):
        """Format Longitude Data Correctly"""
        if self.coord_format == ’dd’:
            decimal_degrees = self._longitude[0] + (self._longitude[1] / 60)
            return [decimal_degrees, self._longitude[2]]
        elif self.coord_format == ’dms’:
            minute_parts = modf(self._longitude[1])
            seconds = round(minute_parts[0] * 60)
            return [self._longitude[0], int(minute_parts[1]), seconds, self._longitude[2]]
        else:
            return self._longitude


    def start_logging(self, target_file, mode="append"):
        """Create GPS data log object"""
        # Set Write Mode Overwrite or Append
        mode_code = ’w’ if mode == ’new’ else ’a’

        try:
            self.log_handle = open(target_file, mode_code)
        except AttributeError:
            print("Invalid FileName")
            return False

        self.log_enlog_en = True
        return True

    def stop_logging(self):
        """Closes the log file handler and disables further logging"""
        try:
            self.log_handle.close()
        except AttributeError:
            print("Invalid Handle")
            return False

        self.log_enlog_en = False
        return True

    def write_log(self, log_string):
        """Attempts to write the last valid NMEA sentence character to the active file handler"""
        try:
            self.log_handle.write(log_string)
        except TypeError:
            return False
        return True

    #########################################
    # Sentence Parsers
    def gprmc(self):
        """Parse Recommended Minimum Specific GPS/Transit data (RMC)Sentence.
        Updates UTC timestamp, latitude, longitude, Course, Speed, Date, and fix status"""

        # UTC Timestamp
        try:
            utc_string = self.gps_segments[1]

            if utc_string:  # Possible timestamp found
                hours = (int(utc_string[0:2]) + self.local_offset) % 24
                minutes = int(utc_string[2:4])
                seconds = float(utc_string[4:])
                self.timestamp = f"{hours:02}:{minutes:02}:{seconds:05.2f}"
            else:  # No Time stamp yet
                self.timestamp = [0,0,0.0]

        except ValueError:  # Bad Timestamp value present
            return False

        # Date stamp
        try:
            date_string = self.gps_segments[9]

            # Date string printer function assumes to be year >=2000,
            # date_string() must be supplied with the correct century argument to display correctly
            if date_string:  # Possible date stamp found
                day = int(date_string[0:2])
                month = int(date_string[2:4])
                year = int(date_string[4:6])
                self.date = (day, month, year)
            else:  # No Date stamp yet
                self.date = (0, 0, 0)

        except ValueError:  # Bad Date stamp value present
            return False

        # Check Receiver Data Valid Flag
        if self.gps_segments[2] == ’A’:  # Data from Receiver is Valid/Has Fix

            # Longitude / Latitude
            try:
                # Latitude
                l_string = self.gps_segments[3]
                lat_degs = int(l_string[0:2])
                lat_mins = float(l_string[2:])
                lat_hemi = self.gps_segments[4]

                # Longitude
                l_string = self.gps_segments[5]
                lon_degs = int(l_string[0:3])
                lon_mins = float(l_string[3:])
                lon_hemi = self.gps_segments[6]
            except ValueError:
                return False

            if lat_hemi not in self.__HEMISPHERES:
                return False

            if lon_hemi not in self.__HEMISPHERES:
                return False

            # Speed
            try:
                spd_knt = float(self.gps_segments[7])
            except ValueError:
                return False

            # Course
            try:
                if self.gps_segments[8]:
                    course = float(self.gps_segments[8])
                else:
                    course = 0.0
            except ValueError:
                return False

            # Update Object Data
            self._latitude = [lat_degs, lat_mins, lat_hemi]
            self._longitude = [lon_degs, lon_mins, lon_hemi]
            # Include mph and hm/h
            self.speed = [spd_knt, spd_knt * 1.151, spd_knt * 1.852]
            self.course = course
            self.valid = True

            # Update Last Fix Time
            self.new_fix_time()

        else:  # Clear Position Data if Sentence is ’Invalid’
            self._latitude = [0, 0.0, ’N’]
            self._longitude = [0, 0.0, ’W’]
            self.speed = [0.0, 0.0, 0.0]
            self.course = 0.0
            self.valid = False

        return True

    def gpgll(self):
        """Parse Geographic Latitude and Longitude (GLL)Sentence. Updates UTC timestamp, latitude, longitude, and fix status"""

        # UTC Timestamp
        try:
            utc_string = self.gps_segments[5]

            if utc_string:  # Possible timestamp found
                hours = (int(utc_string[0:2]) + self.local_offset) % 24
                minutes = int(utc_string[2:4])
                seconds = float(utc_string[4:])
                self.timestamp = f"{hours:02}:{minutes:02}:{seconds:05.2f}"
            else:  # No Time stamp yet
                self.timestamp = [0,0,0.0]

        except ValueError:  # Bad Timestamp value present
            return False

        # Check Receiver Data Valid Flag
        if self.gps_segments[6] == ’A’:  # Data from Receiver is Valid/Has Fix

            # Longitude / Latitude
            try:
                # Latitude
                l_string = self.gps_segments[1]
                lat_degs = int(l_string[0:2])
                lat_mins = float(l_string[2:])
                lat_hemi = self.gps_segments[2]

                # Longitude
                l_string = self.gps_segments[3]
                lon_degs = int(l_string[0:3])
                lon_mins = float(l_string[3:])
                lon_hemi = self.gps_segments[4]
            except ValueError:
                return False

            if lat_hemi not in self.__HEMISPHERES:
                return False

            if lon_hemi not in self.__HEMISPHERES:
                return False

            # Update Object Data
            self._latitude = [lat_degs, lat_mins, lat_hemi]
            self._longitude = [lon_degs, lon_mins, lon_hemi]
            self.valid = True

            # Update Last Fix Time
            self.new_fix_time()

        else:  # Clear Position Data if Sentence is ’Invalid’
            self._latitude = [0, 0.0, ’N’]
            self._longitude = [0, 0.0, ’W’]
            self.valid = False

        return True

    def gpvtg(self):
        """Parse Track Made Good and Ground Speed (VTG) Sentence. Updates speed and course"""
        try:
            course = float(self.gps_segments[1]) if self.gps_segments[1] else 0.0
            spd_knt = float(self.gps_segments[5]) if self.gps_segments[5] else 0.0
        except ValueError:
            return False

        # Include mph and km/h
        self.speed = (spd_knt, spd_knt * 1.151, spd_knt * 1.852)
        self.course = course
        return True

    def 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"""

        try:
            # UTC Timestamp
            utc_string = self.gps_segments[1]

            # Skip timestamp if receiver doesn’t have on yet
            if utc_string:
                hours = (int(utc_string[0:2]) + self.local_offset) % 24
                minutes = int(utc_string[2:4])
                seconds = float(utc_string[4:])
            else:
                hours = 0
                minutes = 0
                seconds = 0.0

            # Number of Satellites in Use
            satellites_in_use = int(self.gps_segments[7])

            # Get Fix Status
            fix_stat = int(self.gps_segments[6])

        except (ValueError, IndexError):
            return False

        try:
            # Horizontal Dilution of Precision
            hdop = float(self.gps_segments[8])
        except (ValueError, IndexError):
            hdop = 0.0

        # Process Location and Speed Data if Fix is GOOD
        if fix_stat:

            # Longitude / Latitude
            try:
                # Latitude
                l_string = self.gps_segments[2]
                lat_degs = int(l_string[0:2])
                lat_mins = float(l_string[2:])
                lat_hemi = self.gps_segments[3]

                # Longitude
                l_string = self.gps_segments[4]
                lon_degs = int(l_string[0:3])
                lon_mins = float(l_string[3:])
                lon_hemi = self.gps_segments[5]
            except ValueError:
                return False

            if lat_hemi not in self.__HEMISPHERES:
                return False

            if lon_hemi not in self.__HEMISPHERES:
                return False

            # Altitude / Height Above Geoid
            try:
                altitude = float(self.gps_segments[9])
                geoid_height = float(self.gps_segments[11])
            except ValueError:
                altitude = 0
                geoid_height = 0

            # Update Object Data
            self._latitude = [lat_degs, lat_mins, lat_hemi]
            self._longitude = [lon_degs, lon_mins, lon_hemi]
            self.altitude = altitude
            self.geoid_height = geoid_height

        # Update Object Data
        self.timestamp = f"{hours:02}:{minutes:02}:{seconds:05.2f}"
        self.satellites_in_use = satellites_in_use
        self.hdop = hdop
        self.fix_stat = fix_stat

        # If Fix is GOOD, update fix timestamp
        if fix_stat:
            self.new_fix_time()

        return True

    def 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"""

        # Fix Type (None,2D or 3D)
        try:
            fix_type = int(self.gps_segments[2])
        except ValueError:
            return False

        # Read All (up to 12) Available PRN Satellite Numbers
        sats_used = []
        for sats in range(12):
            sat_number_str = self.gps_segments[3 + sats]
            if sat_number_str:
                try:
                    sat_number = int(sat_number_str)
                    sats_used.append(sat_number)
                except ValueError:
                    return False
            else:
                break

        # PDOP,HDOP,VDOP
        try:
            pdop = float(self.gps_segments[15])
            hdop = float(self.gps_segments[16])
            vdop = float(self.gps_segments[17])
        except ValueError:
            return False

        # Update Object Data
        self.fix_type = fix_type

        # If Fix is GOOD, update fix timestamp
        if fix_type > self.__NO_FIX:
            self.new_fix_time()

        self.satellites_used = sats_used
        self.hdop = hdop
        self.vdop = vdop
        self.pdop = pdop

        return True

    def 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"""
        try:
            num_sv_sentences = int(self.gps_segments[1])
            current_sv_sentence = int(self.gps_segments[2])
            sats_in_view = int(self.gps_segments[3])
        except ValueError:
            return False

        # Create a blank dict to store all the satellite data from this sentence in:
        # satellite PRN is key, tuple containing telemetry is value
        satellite_dict = dict()

        # Calculate  Number of Satelites to pull data for and thus how many segment positions to read
        if num_sv_sentences == current_sv_sentence:
            # Last sentence may have 1-4 satellites; 5 - 20 positions
            sat_segment_limit = (sats_in_view - ((num_sv_sentences - 1) * 4)) * 5
        else:
            sat_segment_limit = 20  # Non-last sentences have 4 satellites and thus read up to position 20

        # Try to recover data for up to 4 satellites in sentence
        for sats in range(4, sat_segment_limit, 4):

            # If a PRN is present, grab satellite data
            if self.gps_segments[sats]:
                try:
                    sat_id = int(self.gps_segments[sats])
                except (ValueError,IndexError):
                    return False

                try:  # elevation can be null (no value) when not tracking
                    elevation = int(self.gps_segments[sats+1])
                except (ValueError,IndexError):
                    elevation = None

                try:  # azimuth can be null (no value) when not tracking
                    azimuth = int(self.gps_segments[sats+2])
                except (ValueError,IndexError):
                    azimuth = None

                try:  # SNR can be null (no value) when not tracking
                    snr = int(self.gps_segments[sats+3])
                except (ValueError,IndexError):
                    snr = None
            # If no PRN is found, then the sentence has no more satellites to read
            else:
                break

            # Add Satellite Data to Sentence Dict
            satellite_dict[sat_id] = (elevation, azimuth, snr)

        # Update Object Data
        self.total_sv_sentences = num_sv_sentences
        self.last_sv_sentence = current_sv_sentence
        self.satellites_in_view = sats_in_view

        # For a new set of sentences, we either clear out the existing sat data or
        # update it as additional SV sentences are parsed
        if current_sv_sentence == 1:
            self.satellite_data = satellite_dict
        else:
            self.satellite_data.update(satellite_dict)

        return True

    ##########################################
    # Data Stream Handler Functions
    def new_sentence(self):
        """Adjust Object Flags in Preparation for a New Sentence"""
        self.gps_segments = [”]
        self.active_segment = 0
        self.crc_xor = 0
        self.sentence_active = True
        self.process_crc = True
        self.char_count = 0

    def 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"""

        valid_sentence = False

        # Validate new_char is a printable char
        ascii_char = ord(new_char)

        if 10 <= ascii_char <= 126:
            self.char_count += 1

            # Write Character to log file if enabled
            if self.log_en:
                self.write_log(new_char)

            # Check if a new string is starting ($)
            if new_char == ’$’:
                self.new_sentence()
                return None

            elif self.sentence_active:

                # Check if sentence is ending (*)
                if new_char == ’*’:
                    self.process_crc = False
                    self.active_segment += 1
                    self.gps_segments.append(”)
                    return None

                # Check if a section is ended (,), Create a new substring to feed
                # characters to
                elif new_char == ’,’:
                    self.active_segment += 1
                    self.gps_segments.append(”)

                # Store All Other printable character and check CRC when ready
                else:
                    self.gps_segments[self.active_segment] += new_char

                    # When CRC input is disabled, sentence is nearly complete
                    if not self.process_crc:

                        if len(self.gps_segments[self.active_segment]) == 2:
                            try:
                                final_crc = int(self.gps_segments[self.active_segment], 16)
                                if self.crc_xor == final_crc:
                                    valid_sentence = True
                                else:
                                    self.crc_fails += 1
                            except ValueError:
                                pass  # CRC Value was deformed and could not have been correct

                # Update CRC
                if self.process_crc:
                    self.crc_xor ^= ascii_char

                # If a Valid Sentence Was received and it’s a supported sentence, then parse it!!
                if valid_sentence:
                    self.clean_sentences += 1  # Increment clean sentences received
                    self.sentence_active = False  # Clear Active Processing Flag

                    if self.gps_segments[0] in self.supported_sentences:

                        # parse the Sentence Based on the message type, return True if parse is clean
                        if self.supported_sentences[self.gps_segments[0]](self):

                            # Let host know that the GPS object was updated by returning parsed sentence type
                            self.parsed_sentences += 1
                            return self.gps_segments[0]

                # Check that the sentence buffer isn’t filling up with Garage waiting for the sentence to complete
                if self.char_count > self.SENTENCE_LIMIT:
                    self.sentence_active = False

        # Tell Host no new sentence was parsed
        return None

    def 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"""
        try:
            self.fix_time = utime.ticks_ms()
        except NameError:
            self.fix_time = time.time()

    ########################################
    # User Helper Functions working with the GPS object data easier
    def satellite_data_updated(self):
        """Checks if the all the GSV sentences in a group have been read, making satellite data complete:return: boolean"""
        if self.total_sv_sentences > 0 and self.total_sv_sentences == self.last_sv_sentence:
            return True
        else:
            return False

    def unset_satellite_data_updated(self):
        """Mark GSV sentences as read indicating the data has been used and future updates are fresh"""
        self.last_sv_sentence = 0

    def satellites_visible(self):
        """Returns a list of of the satellite PRNs currently visible to the receiver:return: list"""
        return list(self.satellite_data.keys())

    def time_since_fix(self):
        """Returns number of millisecond since the last sentence with a valid fix was parsed. Returns 0 if no fix has been found"""
        # Test if a Fix has been found
        if self.fix_time == 0:
            return -1

        # Try calculating fix time using utime; if not running MicroPython
        # time.time() returns a floating point value in secs
        try:
            current = utime.ticks_diff(utime.ticks_ms(), self.fix_time)
        except NameError:
            current = (time.time() - self.fix_time) * 1000  # ms

        return current

    def compass_direction(self):
        """Determine a cardinal or inter-cardinal direction based on current course.:return: string"""
        # Calculate the offset for a rotated compass
        if self.course >= 348.75:
            offset_course = 360 - self.course
        else:
            offset_course = self.course + 11.25

        # Each compass point is separated by 22.5 degrees, divide to find lookup value
        dir_index = floor(offset_course / 22.5)

        final_dir = self.__DIRECTIONS[dir_index]

        return final_dir

    def latitude_string(self):
        """Create a readable string of the current latitude data:return: string"""
        if self.coord_format == ’dd’:
            formatted_latitude = self.latitude
            lat_string = str(formatted_latitude[0]) + ’g ’ + str(self._latitude[2])
        elif self.coord_format == ’dms’:
            formatted_latitude = self.latitude
            lat_string = str(formatted_latitude[0]) + ’g ’ + str(formatted_latitude[1]) + "’ " + str(formatted_latitude[2]) + ’" ’ + str(formatted_latitude[3])
        else:
            lat_string = str(self._latitude[0]) + ’g ’ + str(self._latitude[1]) + "’ " + str(self._latitude[2])
        return lat_string

    def longitude_string(self):
        """Create a readable string of the current longitude data: return: string"""
        if self.coord_format == ’dd’:
            formatted_longitude = self.longitude
            lon_string = str(formatted_longitude[0]) + ’g ’ + str(self._longitude[2])
        elif self.coord_format == ’dms’:
            formatted_longitude = self.longitude
            lon_string = str(formatted_longitude[0]) + ’g ’ + str(formatted_longitude[1]) + "’ " + str(formatted_longitude[2]) + ’" ’ + str(formatted_longitude[3])
        else:
            lon_string = str(self._longitude[0]) + ’g ’ + str(self._longitude[1]) + "’ " + str(self._longitude[2])
        return lon_string

    def 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:
        """
        if unit == ’mph’:
            speed_string = f"{self.speed[1]:.2f} mph"

        else:
            speed_string = f"{self.speed[2]:.2f} km/h"

        return speed_string

    def 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
        """

        # Long Format Januray 1st, 2014
        if formatting == ’long’:
            # Retrieve Month string from private set
            month = self.__MONTHS[self.date[1] - 1]

            # Determine Date Suffix
            if self.date[0] in (1, 21, 31):
                suffix = ’st’
            elif self.date[0] in (2, 22):
                suffix = ’nd’
            elif self.date[0] == (3, 23):
                suffix = ’rd’
            else:
                suffix = ’th’

            day = str(self.date[0]) + suffix  # Create Day String

            year = century + str(self.date[2])  # Create Year String

            date_string = month + ’ ’ + day + ’, ’ + year  # Put it all together

        else:
            # Add leading zeros to day string if necessary
            if self.date[0] < 10:
                day = ’0’ + str(self.date[0])
            else:
                day = str(self.date[0])

            # Add leading zeros to month string if necessary
            if self.date[1] < 10:
                month = ’0’ + str(self.date[1])
            else:
                month = str(self.date[1])

            # Add leading zeros to year string if necessary
            if self.date[2] < 10:
                year = ’0’ + str(self.date[2])
            else:
                year = str(self.date[2])

            # Build final string based on desired formatting
            if formatting == ’s_dmy’:
                date_string = day + ’/’ + month + ’/’ + year

            else:  # Default date format
                date_string = month + ’/’ + day + ’/’ + year

        return date_string

    # All the currently supported NMEA sentences
    supported_sentences = {’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,
                          }

if __name__ == "__main__":
    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.

```
"""
===============================================================================
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

===============================================================================
"""

import machine
from machine import I2C, Pin
from sh1106 import SH1106_I2C
from time import sleep
from libGPS import MicropyGPS

# Constants
I2C_SCL_PIN = 22  # Pin for I2C clock line
I2C_SDA_PIN = 21  # Pin for I2C data line
I2C_FREQ = 100_000  # Frequency for I2C communication in Hz
UART_TX_PIN = 17  # Pin for UART TX
UART_RX_PIN = 16  # Pin for UART RX
UART_BAUDRATE = 9600  # Baud rate for UART communication
OLED_CONTRAST = 80  # Contrast setting for the OLED display (0-255)
DISPLAY_REFRESH_INTERVAL = 1  # Interval for refreshing the display in seconds

# Function: initialize_display
# Description: Initializes the OLED display via I2C.
# Returns: The initialized OLED object.
# Raises: Exception if the I2C or OLED initialization fails.
def initialize_display():
    """
    Set up the I2C bus and OLED display.
    """
    try:
        # Initialize I2C bus
        i2c = I2C(0, scl=Pin(I2C_SCL_PIN), sda=Pin(I2C_SDA_PIN), freq=I2C_FREQ)
        # Initialize OLED display
        oled = SH1106_I2C(i2c)
        oled.contrast(OLED_CONTRAST)
        return oled
    except Exception as e:
        print(f"Error initializing OLED: {e}")
        raise

# Function: update_display
# Description: Updates the OLED display with the given GPS data.
# Parameters:
#   oled (SH1106_I2C): The OLED display object.
#   gps_data (dict): Dictionary containing GPS data (time, date, lat, long, etc.).
def update_display(oled, gps_data):
    """
    Refreshes the OLED display with new GPS data.
    """
    oled.fill(0)  # Clear the screen

    # Display labels
    oled.text("Date:", 0, 0) #(text, x position, y position)
    oled.text("Time:", 0, 9)
    oled.text("Lat:", 0, 18)
    oled.text("Long:", 0, 27)
    oled.text("Altitude:", 0, 36)
    oled.text("Speed:", 0, 45)
    oled.text("N Satellites:", 0, 54)

    # Display GPS data
    oled.text(f"{gps_data[’date’]}", 38, 0)  # Truncate time to HH:MM:SS
    oled.text(f"{gps_data[’time’]}", 38, 9)
    oled.text(f"{gps_data[’latitude’][:12]}", 30, 18)
    oled.text(f"{gps_data[’longitude’][:12]}", 38, 27)
    oled.text(f"{gps_data[’altitude’]} m", 70, 36)
    oled.text(f"{gps_data[’speed’]}", 47, 45)
    oled.text(f"{gps_data[’satellites’]}", 103, 54)
    oled.show()  # Update the OLED screen


# Function: parse_gps_data
# Description: Parses raw GPS data from the MicropyGPS object into a dictionary.
# Parameters:
#   my_gps (MicropyGPS): The MicropyGPS object handling GPS data parsing.
# Returns: Dictionary containing parsed GPS data.
def parse_gps_data(my_gps):
    """
    Extracts GPS data from the MicropyGPS object and formats it.
    """
    return {
        ’date’: my_gps.date_string(’s_dmy’),  # Date in DD/MM/YYYY format
        ’time’: my_gps.timestamp,  # Format time as HH:MM:SS
        ’latitude’: my_gps.latitude_string(),  # Latitude in degrees/minutes/seconds
        ’longitude’: my_gps.longitude_string(),  # Longitude in degrees/minutes/seconds
        ’altitude’: str(my_gps.altitude),  # Altitude in meters
        ’speed’: my_gps.speed_string(),  # Longitude in degrees/minutes/seconds
        ’satellites’: str(my_gps.satellites_in_use),  # Number of satellites in use
        ’pdop’: str(my_gps.pdop),  # Horizontal Dilution of Precision
        ’hdop’: str(my_gps.hdop),  # Horizontal Dilution of Precision
        ’vdop’: str(my_gps.vdop),  # Horizontal Dilution of Precision
    }

# Function: main
# Description: Main function to initialize components and handle GPS data processing and display updates.
def main():
    """
    The main program loop. Initializes the OLED display and GPS module, then continuously
    updates the display with parsed GPS data.
    """
    # Initialize the OLED display
    oled = initialize_display()

    # Instantiate the GPS parser
    my_gps = MicropyGPS()

    # Initialize UART for GPS communication
    gps_serial = machine.UART(2, baudrate=UART_BAUDRATE, tx=UART_TX_PIN, rx=UART_RX_PIN)

    print("System initialized. Press Ctrl+C to exit.")
    try:
        while True:
            gps_data_available = False
            # Read GPS data from the UART interface
            while gps_serial.any():
                data = gps_serial.read()
                for byte in data:
                    # Update GPS parser with each byte
                    if my_gps.update(chr(byte)) is not None:
                        gps_data_available = True

            # If new GPS data is available, process and display it
            if gps_data_available:
                gps_data = parse_gps_data(my_gps)
                update_display(oled, gps_data)

                # Print GPS data line by line for readability
                print("\n--- GPS Data ---")
                for key, value in gps_data.items():
                    # Print GPS data line by line for readability
                    print("\n--- GPS Data ---")
                    print(f"Date: {gps_data[’date’]}")
                    print(f"Time: {gps_data[’time’]}")
                    print(f"Latitude: {gps_data[’latitude’]}")
                    print(f"Longitude: {gps_data[’longitude’]}")
                    print(f"Altitude: {gps_data[’altitude’]} meters")
                    print(f"Speed: {gps_data[’speed’]}")
                    print(f"Satellites: {gps_data[’satellites’]}")
                    print(f"Position Dilution of Precision: {gps_data[’pdop’]}")
                    print(f"Horizontal Dilution of Precision: {gps_data[’hdop’]}")
                    print(f"Vertical Dilution of Precision: {gps_data[’vdop’]}")
                print("-----------------")

            sleep(DISPLAY_REFRESH_INTERVAL)  # Delay to control refresh rate
    except KeyboardInterrupt:
        print("Program stopped. Exiting...")
        oled.poweroff()  # Turn off the OLED display

# Entry point for the script
if __name__ == "__main__":
    main()
```

7.5 Code/test.py File Reference

Namespaces

- namespace test

7.6 test.py

Go to the documentation of this file.