KubiSat Firmware

Generated by Doxygen 1.13.2

Chapter 1

Clock Commands

Member handle_clock_sync_interval (const std::string &param, OperationType operationType)

Command ID: 3.3

Member handle_enable_gps_uart_passthrough (const std::string &param, OperationType operationType)

Command ID: 7.2

Member handle_enter_bootloader_mode (const std::string &param, OperationType operationType)

Command ID: 2

Member handle_get_build_version (const std::string &param, OperationType operationType)

Command ID: 1

Member handle_get_commands_list (const std::string &param, OperationType operationType)

Command ID: 0

Member handle_get_current_charge_solar (const std::string &param, OperationType operationType)

Command ID: 2.5

Member handle_get_current_charge_total (const std::string &param, OperationType operationType)

Command ID: 2.6

Member handle_get_current_charge_usb (const std::string &param, OperationType operationType)

Command ID: 2.4

Member handle_get_current_draw (const std::string &param, OperationType operationType)

Command ID: 2.7

Member handle_get_event_count (const std::string &param, OperationType operationType)

Command ID: 5.2

Member handle_get_gga_data (const std::string &param, OperationType operationType)

Command ID: 7.4

Member handle_get_last_events (const std::string &param, OperationType operationType)

Command ID: 5.1

Member handle_get_last_sync_time (const std::string &param, OperationType operationType)

Command ID: 3.7

Member handle_get_last_telemetry_record (const std::string &param, OperationType operationType)

Command ID: 8.2

Member handle_get_power_manager_ids (const std::string &param, OperationType operationType)

Command ID: 2.0

Member handle_get_rmc_data (const std::string &param, OperationType operationType)

Command ID: 7.3

Member handle_get_sensor_data (const std::string &param, OperationType operationType)

Command ID: 3.0

Member handle_get_sensor_list (const std::string &param, OperationType operationType)

Command ID: 4.2

Member handle_get_voltage_5v (const std::string &param, OperationType operationType)

Command ID: 2.3

Member handle_get_voltage_battery (const std::string &param, OperationType operationType)

Command ID: 2.2

Member handle_gps_power_status (const std::string &param, OperationType operationType)

Command ID: 7.1

Member handle_list_files (const std::string &param, OperationType operationType)

Command ID: 6.0

Member handle_mount (const std::string &param, OperationType operationType)

Command ID: 6.4

Member handle_sensor_config (const std::string &param, OperationType operationType)

Command ID: 3.1

Member handle_time (const std::string &param, OperationType operationType)

Command ID: 3.0

Member handle_timezone_offset (const std::string &param, OperationType operationType)

Command ID: 3.1

Member handle_verbosity (const std::string &param, OperationType operationType)

Command ID: 1.8

Chapter 2

Topic Index

2.1 Topics

Here is a list of all topics with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Topic Documentation

6.1 Clock Management Commands

Commands for managing system time and clock settings.

Functions

- std::vector< Frame > handle_time (const std::string &param, OperationType operationType)
  Handler for getting and setting system time.
- std::vector< Frame > handle_timezone_offset (const std::string &param, OperationType operationType)
  Handler for getting and setting timezone offset.
- std::vector< Frame > handle_clock_sync_interval (const std::string &param, OperationType operationType)
  Handler for getting and setting clock synchronization interval.
- std::vector< Frame > handle_get_last_sync_time (const std::string &param, OperationType operationType)
  Handler for getting last clock sync time.

6.1.1 Detailed Description

Commands for managing system time and clock settings.

6.1.2 Function Documentation

6.1.2.1 handle_time()

```
std::vector< Frame > handle_time (
            const std::string & param,
            OperationType operationType)
```

Handler for getting and setting system time.

Parameters

param	For SET: Unix timestamp as string, for GET: empty string
operationType	GET/SET

Returns

Vector of frames containing success/error and current time or confirmation

Note

GET: **KBST;0;GET;3;0;;KBST**

When getting time, returns format "HH:MM:SS Weekday DD.MM.YYYY"

SET: **KBST;0;SET;3;0;TIMESTAMP;KBST**

When setting time, expects Unix timestamp as parameter

Command Command ID: 3.0

Definition at line 31 of file clock_commands.cpp.

6.1.2.2 handle_timezone_offset()

```
std::vector< Frame > handle_timezone_offset (
            const std::string & param,
            OperationType operationType)
```

Handler for getting and setting timezone offset.

Parameters

param	For SET: Timezone offset in minutes (-720 to +720), for GET: empty string
operationType	GET/SET

Returns

Vector of frames containing success/error and timezone offset in minutes

Note

GET: **KBST;0;GET;3;1;;KBST**

SET: **KBST;0;SET;3;1;OFFSET;KBST**

Command Command ID: 3.1

Definition at line 96 of file clock_commands.cpp.

6.1.2.3 handle_clock_sync_interval()

```
std::vector< Frame > handle_clock_sync_interval (
            const std::string & param,
            OperationType operationType)
```

Handler for getting and setting clock synchronization interval.

Parameters

param	For SET: Sync interval in seconds, for GET: empty string
operationType	GET/SET

Returns

Vector with frame containing success/error and sync interval in seconds

Note

GET: **KBST;0;GET;3;3;;KBST**

SET: **KBST;0;SET;3;3;INTERVAL;KBST**

Command Command ID: 3.3

Definition at line 155 of file clock_commands.cpp.

6.1.2.4 handle_get_last_sync_time()

```
std::vector< Frame > handle_get_last_sync_time (
            const std::string & param,
            OperationType operationType)
```

Handler for getting last clock sync time.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector with one frame containing success/error and last sync time as Unix timestamp

Note

KBST;0;GET;3;7;;KBST

Command Command ID: 3.7

Definition at line 211 of file clock_commands.cpp.

6.2 Command System

Core command system implementation.

Typedefs

- using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>
  Function type for command handlers.
- using CommandMap = std::map<uint32_t, CommandHandler>
  Map type for storing command handlers.

Functions

- std::vector< Frame > execute_command (uint32_t commandKey, const std::string &param, OperationType operationType)
  Executes a command based on its key.

Variables

- CommandMap command_handlers
  Global map of all command handlers.

6.2.1 Detailed Description

Core command system implementation.

6.2.2 Typedef Documentation

6.2.2.1 CommandHandler

```
using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>
```

Function type for command handlers.

Definition at line 15 of file commands.cpp.

6.2.2.2 CommandMap

```
using CommandMap = std::map<uint32_t, CommandHandler>
```

Map type for storing command handlers.

Definition at line 21 of file commands.cpp.

6.2.3 Function Documentation

6.2.3.1 execute_command()

```
std::vector< Frame > execute_command (
            uint32_t commandKey,
            const std::string & param,
            OperationType operationType)
```

Executes a command based on its key.

Parameters

commandKey	Combined group and command ID (group << 8 | command)
param	Command parameter string
operationType	Operation type (GET/SET)

Returns

Frame Response frame containing execution result

Looks up the command handler in commandHandlers map and executes it

Definition at line 67 of file commands.cpp.

6.2.4 Variable Documentation

6.2.4.1 command_handlers

```
CommandMap command_handlers
```

Initial value:

```
= {
    {((static_cast<uint32_t>(1) « 8) | static_cast<uint32_t>(0)), handle_get_commands_list},
    {((static_cast<uint32_t>(1) « 8) | static_cast<uint32_t>(1)), handle_get_build_version},
    {((static_cast<uint32_t>(1) « 8) | static_cast<uint32_t>(8)), handle_verbosity},
    {((static_cast<uint32_t>(1) « 8) | static_cast<uint32_t>(9)), handle_enter_bootloader_mode},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(0)), handle_get_power_manager_ids},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(2)), handle_get_voltage_battery},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(3)), handle_get_voltage_5v},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(4)), handle_get_current_charge_usb},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(5)), handle_get_current_charge_solar},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(6)), handle_get_current_charge_total},
    {((static_cast<uint32_t>(2) « 8) | static_cast<uint32_t>(7)), handle_get_current_draw},
    {((static_cast<uint32_t>(3) « 8) | static_cast<uint32_t>(0)), handle_time},
    {((static_cast<uint32_t>(3) « 8) | static_cast<uint32_t>(1)), handle_timezone_offset},
    {((static_cast<uint32_t>(3) « 8) | static_cast<uint32_t>(2)), handle_clock_sync_interval},
    {((static_cast<uint32_t>(3) « 8) | static_cast<uint32_t>(3)), handle_get_last_sync_time},
    {((static_cast<uint32_t>(4) « 8) | static_cast<uint32_t>(0)), handle_get_sensor_data},
    {((static_cast<uint32_t>(4) « 8) | static_cast<uint32_t>(1)), handle_sensor_config},
    {((static_cast<uint32_t>(4) « 8) | static_cast<uint32_t>(3)), handle_get_sensor_list},
    {((static_cast<uint32_t>(5) « 8) | static_cast<uint32_t>(1)), handle_get_last_events},
    {((static_cast<uint32_t>(5) « 8) | static_cast<uint32_t>(2)), handle_get_event_count},
    {((static_cast<uint32_t>(6) « 8) | static_cast<uint32_t>(0)), handle_list_files},
    {((static_cast<uint32_t>(6) « 8) | static_cast<uint32_t>(4)), handle_mount},
    {((static_cast<uint32_t>(7) « 8) | static_cast<uint32_t>(1)), handle_gps_power_status},
    {((static_cast<uint32_t>(7) « 8) | static_cast<uint32_t>(2)), handle_enable_gps_uart_passthrough},
    {((static_cast<uint32_t>(7) « 8) | static_cast<uint32_t>(3)), handle_get_rmc_data},
    {((static_cast<uint32_t>(7) « 8) | static_cast<uint32_t>(4)), handle_get_gga_data},
    {((static_cast<uint32_t>(8) « 8) | static_cast<uint32_t>(2)), handle_get_last_telemetry_record},
    {((static_cast<uint32_t>(8) « 8) | static_cast<uint32_t>(3)), handle_get_last_sensor_record},
}
```

Global map of all command handlers.

Maps command keys (group << 8 | command) to their handler functions

Definition at line 27 of file commands.cpp.

6.3 Diagnostic Commands

Functions

- std::vector< Frame > handle_get_commands_list (const std::string &param, OperationType operationType)

    Handler for listing all available commands on UART.
- std::vector< Frame > handle_get_build_version (const std::string &param, OperationType operationType)

    Get firmware build version.
- std::vector< Frame > handle_verbosity (const std::string &param, OperationType operationType)

    Handles setting or getting the UART verbosity level.
- std::vector< Frame > handle_enter_bootloader_mode (const std::string &param, OperationType operation←Type)

    Reboot system to USB firmware loader.

6.3.1 Detailed Description

6.3.2 Function Documentation

6.3.2.1 handle_get_commands_list()

```
std::vector< Frame > handle_get_commands_list (
            const std::string & param,
            OperationType operationType)
```

Handler for listing all available commands on UART.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of response frames - start frame, sequence of elements, end frame

Note

KBST;0;GET;1;0;;TSBK

Print all available commands on UART port

Command Command ID: 0

Definition at line 21 of file diagnostic_commands.cpp.

6.3.2.2 handle_get_build_version()

```
std::vector< Frame > handle_get_build_version (
            const std::string & param,
            OperationType operationType)
```

Get firmware build version.

Parameters

param	Empty string expected
operationType	GET

Returns

One-element vector with result frame

Note

KBST;0;GET;1;1;;TSBK

Get the firmware build version

Command Command ID: 1

Definition at line 75 of file diagnostic_commands.cpp.

6.3.2.3 handle_verbosity()

```
std::vector< Frame > handle_verbosity (
            const std::string & param,
            OperationType operationType)
```

Handles setting or getting the UART verbosity level.

This function allows the user to either retrieve the current UART verbosity level or set a new verbosity level.

Parameters

param	The desired verbosity level (0-5) as a string. If empty, the current level is returned.
operationType	The operation type. Must be GET to retrieve the current level, or SET to set a new level.

Returns

Vector containing one frame indicating the result of the operation.

- Success (GET): Frame containing the current verbosity level.
- Success (SET): Frame with "LEVEL SET" message.
- Error: Frame with error message (e.g., "INVALID LEVEL (0-5)", "INVALID FORMAT").

Note

KBST;0;GET;1;8;;TSBK - Gets the current verbosity level.

KBST;0;SET;1;8;[level];TSBK - Sets the verbosity level.

Example: **KBST;0;SET;1;8;2;TSBK** - Sets the verbosity level to 2.

Command Command ID: 1.8

Definition at line 117 of file diagnostic_commands.cpp.

6.3.2.4 handle_enter_bootloader_mode()

```
std::vector< Frame > handle_enter_bootloader_mode (
            const std::string & param,
            OperationType operationType)
```

Reboot system to USB firmware loader.

Parameters

param	Empty string expected
operationType	Must be SET

Returns

Frame with operation result

Note

KBST;0;SET;1;9;;TSBK

Reboot the system to USB firmware loader

Command Command ID: 2

Definition at line 157 of file diagnostic_commands.cpp.

6.4 Event Commands

Commands for accessing and managing system event logs.

Functions

- std::vector< Frame > handle_get_last_events (const std::string &param, OperationType operationType)

  Handler for retrieving last N events from the event log.
- std::vector< Frame > handle_get_event_count (const std::string &param, OperationType operationType)

  Handler for getting total number of events in the log.

6.4.1 Detailed Description

Commands for accessing and managing system event logs.

6.4.2 Function Documentation

6.4.2.1 handle_get_last_events()

```
std::vector< Frame > handle_get_last_events (
            const std::string & param,
            OperationType operationType)
```

Handler for retrieving last N events from the event log.

Parameters

param	Number of events to retrieve (optional, default 10). If 0, all events are returned.
operationType	GET

Returns

Frame containing:

- Success: A sequence of frames, each containing up to 10 hex-encoded events. Each event is in the format IIIITTTTTTTTGGEE, separated by '-'.
  - IIII: Event ID (16-bit, 4 hex characters)
  - TTTTTTTT: Unix Timestamp (32-bit, 8 hex characters)
  - GG: Event Group (8-bit, 2 hex characters)
  - EE: Event Type (8-bit, 2 hex characters) The last frame in the sequence is a VAL frame with the message "SEQ_DONE".
- Error: A single frame with an error message:
  - "INVALID OPERATION": If the operation type is not GET.
  - "INVALID COUNT": If the count is greater than EVENT_BUFFER_SIZE.
  - "INVALID PARAMETER": If the parameter is not a valid unsigned integer.

Note

KBST;0;GET;5;1;[N];TSBK - Retrieves the last N events. If N is 0, retrieves all events.

Returns up to 10 most recent events per frame.

Command Command ID: 5.1

Definition at line 33 of file event_commands.cpp.

6.4.2.2 handle_get_event_count()

```
std::vector< Frame > handle_get_event_count (
            const std::string & param,
            OperationType operationType)
```

Handler for getting total number of events in the log.

Parameters

param	Empty string expected
operationType	GET

Returns

Frame containing:

- Success: Number of events currently in the log
- Error: "INVALID REQUEST"

Note

KBST;0;GET;5;2;;TSBK

Returns the total number of events in the log

Command Command ID: 5.2

Definition at line 101 of file event_commands.cpp.

6.5 GPS Commands

Commands for controlling and monitoring the GPS module.

Functions

- std::vector< Frame > handle_gps_power_status (const std::string &param, OperationType operationType)

  Handler for controlling GPS module power state.
- std::vector< Frame > handle_enable_gps_uart_passthrough (const std::string &param, OperationType operationType)

  Handler for enabling GPS transparent mode (UART pass-through)
- std::vector< Frame > handle_get_rmc_data (const std::string &param, OperationType operationType)

  Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.
- std::vector< Frame > handle_get_gga_data (const std::string &param, OperationType operationType)

  Handler for retrieving GPS GGA (Global Positioning System Fix Data) data.

6.5.1 Detailed Description

Commands for controlling and monitoring the GPS module.

6.5.2 Function Documentation

6.5.2.1 handle_gps_power_status()

```
std::vector< Frame > handle_gps_power_status (
            const std::string & param,
            OperationType operationType)
```

Handler for controlling GPS module power state.

Parameters

param	For SET: "0" to power off, "1" to power on. For GET: empty
operationType	GET to read current state, SET to change state

Returns

Vector of Frames containing:

- Success: Current power state (0/1) or
- Error: Error reason

Note

KBST;0;GET;7;1;;TSBK

Return current GPS module power state: ON/OFF

KBST;0;SET;7;1;POWER;TSBK

POWER - 0 - OFF, 1 - ON

Command Command ID: 7.1

Definition at line 33 of file gps_commands.cpp.

6.5.2.2 handle_enable_gps_uart_passthrough()

```
std::vector< Frame > handle_enable_gps_uart_passthrough (
            const std::string & param,
            OperationType operationType)
```

Handler for enabling GPS transparent mode (UART pass-through)

Parameters

param	TIMEOUT in seconds (optional, defaults to 60)
operationType	SET

Returns

Vector of Frames containing:

- Success: Exit message + reason or
- Error: Error reason

Note

KBST;0;SET;7;2;TIMEOUT;TSBK

TIMEOUT - 1-600s, default 60s

Enters a pass-through mode where UART communication is bridged directly to GPS

Send "##EXIT##" to exit mode before TIMEOUT

Command Command ID: 7.2

Definition at line 90 of file gps_commands.cpp.

6.5.2.3 handle_get_rmc_data()

```
std::vector< Frame > handle_get_rmc_data (
            const std::string & param,
            OperationType operationType)
```

Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: Comma-separated RMC tokens or
- Error: Error message

Note

KBST;0;GET;7;3;;TSBK

Command Command ID: 7.3

Definition at line 193 of file gps_commands.cpp.

6.5.2.4 handle_get_gga_data()

```
std::vector< Frame > handle_get_gga_data (
            const std::string & param,
            OperationType operationType)
```

Handler for retrieving GPS GGA (Global Positioning System Fix Data) data.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: Comma-separated GGA tokens or
- Error: Error message

Note

KBST;0;GET;7;4;;TSBK

Command Command ID: 7.4

Definition at line 236 of file gps_commands.cpp.

6.6 Power Commands

Commands for monitoring power subsystem and battery management.

Functions

- std::vector< Frame > handle_get_power_manager_ids (const std::string &param, OperationType operationType)

  Handler for retrieving Power Manager IDs.
- std::vector< Frame > handle_get_voltage_battery (const std::string &param, OperationType operationType)

  Handler for getting battery voltage.
- std::vector< Frame > handle_get_voltage_5v (const std::string &param, OperationType operationType)

  Handler for getting 5V rail voltage.
- std::vector< Frame > handle_get_current_charge_usb (const std::string &param, OperationType operationType)

  Handler for getting USB charge current.
- std::vector< Frame > handle_get_current_charge_solar (const std::string &param, OperationType operationType)

  Handler for getting solar panel charge current.
- std::vector< Frame > handle_get_current_charge_total (const std::string &param, OperationType operationType)

  Handler for getting total charge current.
- std::vector< Frame > handle_get_current_draw (const std::string &param, OperationType operationType)

  Handler for getting system current draw.

6.6.1 Detailed Description

Commands for monitoring power subsystem and battery management.

6.6.2 Function Documentation

6.6.2.1 handle_get_power_manager_ids()

```
std::vector< Frame > handle_get_power_manager_ids (
            const std::string & param,
            OperationType operationType)
```

Handler for retrieving Power Manager IDs.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: String of Power Manager IDs
- Error: Error message

Note

KBST;0;GET;2;0;;TSBK

This command is used to retrieve the IDs of the Power Manager

Command Command ID: 2.0

Definition at line 30 of file power_commands.cpp.

6.6.2.2 handle_get_voltage_battery()

```
std::vector< Frame > handle_get_voltage_battery (
            const std::string & param,
            OperationType operationType)
```

Handler for getting battery voltage.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: Battery voltage in Volts
- Error: Error message

Note

KBST;0;GET;2;2;;TSBK

This command is used to retrieve the battery voltage

Command Command ID: 2.2

Definition at line 63 of file power_commands.cpp.

6.6.2.3 handle_get_voltage_5v()

```
std::vector< Frame > handle_get_voltage_5v (
            const std::string & param,
            OperationType operationType)
```

Handler for getting 5V rail voltage.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: 5V rail voltage in Volts
- Error: Error message

Note

KBST;0;GET;2;3;;TSBK

This command is used to retrieve the 5V rail voltage

Command Command ID: 2.3

Definition at line 96 of file power_commands.cpp.

6.6.2.4 handle_get_current_charge_usb()

```
std::vector< Frame > handle_get_current_charge_usb (
            const std::string & param,
            OperationType operationType)
```

Handler for getting USB charge current.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: USB charge current in milliamps
- Error: Error message

Note

KBST;0;GET;2;4;;TSBK

This command is used to retrieve the USB charge current

Command Command ID: 2.4

Definition at line 129 of file power_commands.cpp.

6.6.2.5 handle_get_current_charge_solar()

```
std::vector< Frame > handle_get_current_charge_solar (
            const std::string & param,
            OperationType operationType)
```

Handler for getting solar panel charge current.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: Solar charge current in milliamps
- Error: Error message

Note

KBST;0;GET;2;5;;TSBK

This command is used to retrieve the solar panel charge current

Command Command ID: 2.5

Definition at line 162 of file power_commands.cpp.

6.6.2.6 handle_get_current_charge_total()

```
std::vector< Frame > handle_get_current_charge_total (
            const std::string & param,
            OperationType operationType)
```

Handler for getting total charge current.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: Total charge current (USB + Solar) in milliamps
- Error: Error message

Note

KBST;0;GET;2;6;;TSBK

This command is used to retrieve the total charge current

Command Command ID: 2.6

Definition at line 195 of file power_commands.cpp.

6.6.2.7 handle_get_current_draw()

```
std::vector< Frame > handle_get_current_draw (
            const std::string & param,
            OperationType operationType)
```

Handler for getting system current draw.

Parameters

param	Empty string expected
operationType	GET

Returns

Vector of Frames containing:

- Success: System current consumption in milliamps
- Error: Error message

Note

KBST;0;GET;2;7;;TSBK

This command is used to retrieve the system current draw

Command Command ID: 2.7

Definition at line 228 of file power_commands.cpp.

6.7 Sensor Commands

Commands for reading and configuring sensors.

Functions

- std::vector< Frame > handle_get_sensor_data (const std::string &param, OperationType operationType)
  Handler for reading sensor data.
- std::vector< Frame > handle_sensor_config (const std::string &param, OperationType operationType)
  Handler for configuring sensors.
- std::vector< Frame > handle_get_sensor_list (const std::string &param, OperationType operationType)
  Handler for listing available sensors.

6.7.1 Detailed Description

Commands for reading and configuring sensors.

6.7.2 Function Documentation

6.7.2.1 handle_get_sensor_data()

```
std::vector< Frame > handle_get_sensor_data (
            const std::string & param,
            OperationType operationType)
```

Handler for reading sensor data.

Parameters

param	String in format "sensor_type[-data_type]" where: • sensor_type: "light", "environment", "magnetometer", "imu" • data_type (optional): specific data type for the sensor – For light: "light_level" – For environment: "temperature", "pressure", "humidity"
operationType	GET

Returns

Vector of Frames containing:

- Success: Sensor data value(s)
- Error: Error message

Note

KBST;0;GET;3;0;light-light_level;TSBK

This command is used to read data from sensors

Command Command ID: 3.0

Definition at line 34 of file sensor_commands.cpp.

6.7.2.2 handle_sensor_config()

```
std::vector< Frame > handle_sensor_config (
            const std::string & param,
            OperationType operationType)
```

Handler for configuring sensors.

Parameters

param	String in format "sensor_type;key1:value1|key2:value2|..." • sensor_type: "light", "environment", "magnetometer", "imu" • key-value pairs for configuration parameters
operationType	SET

Returns

Vector of Frames containing:

- Success: Success message
- Error: Error message

Note

KBST;0;SET;3;1;light;measurement_mode:continuously_high_resolution;TSBK

This command is used to configure sensors

Command Command ID: 3.1

Definition at line 172 of file sensor_commands.cpp.

6.7.2.3 handle_get_sensor_list()

```
std::vector< Frame > handle_get_sensor_list (
            const std::string & param,
            OperationType operationType)
```

Handler for listing available sensors.

Parameters

param	Empty string or optional filter criteria
operationType	GET

Returns

Vector of Frames containing:

- Success: List of available sensors
- Error: Error message

Note

KBST;0;GET;4;2;;TSBK (lists all sensors)

This command is used to get a list of available sensors

Command Command ID: 4.2

Definition at line 252 of file sensor_commands.cpp.

6.8 Storage Commands

Commands for interacting with the SD card storage.

Functions

- std::vector< Frame > handle_list_files (const std::string &param, OperationType operationType)

  Handles the list files command.
- std::vector< Frame > handle_mount (const std::string &param, OperationType operationType)

  Handles the SD card mount/unmount command.

6.8.1 Detailed Description

Commands for interacting with the SD card storage.

6.8.2 Function Documentation

6.8.2.1 handle_list_files()

```
std::vector< Frame > handle_list_files (
            const std::string & param,
            OperationType operationType)
```

Handles the list files command.

This function lists the files in the root directory of the SD card and sends the filename and size of each file to the ground station.

Parameters

param	Unused.
operationType	The operation type (must be GET).

Returns

A vector of Frames indicating the result of the operation.

- Success: Frame with "File listing complete" message.
- Error: Frame with error message (e.g., "Could not open directory").

Note

KBST;0;GET;6;0;;TSBK

This command lists the files and their sizes in the root directory of the SD card.

Command Command ID: 6.0

Definition at line 37 of file storage_commands.cpp.

6.8.2.2 handle_mount()

```
std::vector< Frame > handle_mount (
            const std::string & param,
            OperationType operationType)
```

Handles the SD card mount/unmount command.

This function mounts or unmounts the SD card.

Parameters

param	"0" to unmount, "1" to mount.
operationType	The operation type (must be SET).

Returns

A vector of Frames indicating the result of the operation.

- Success: Frame with "SD card mounted" or "SD card unmounted" message.
- Error: Frame with error message (e.g., "Invalid parameter", "Mount failed", "Unmount failed").

Note

KBST;0;SET;6;4;[0|1];TSBK

Example: **KBST;0;SET;6;4;1;TSBK** - Mounts the SD card.

Command Command ID: 6.4

Definition at line 126 of file storage_commands.cpp.

6.9 Telemetry Buffer Commands

Commands for interacting with the telemetry buffer.

Functions

- std::vector< Frame > handle_get_last_telemetry_record (const std::string &param, OperationType operationType)

  Handles the get last record command.
- std::vector< Frame > handle_get_last_sensor_record (const std::string &param, OperationType operation←Type)

  Handles the get last sensor record command.

6.9.1 Detailed Description

Commands for interacting with the telemetry buffer.

6.9.2 Function Documentation

6.9.2.1 handle_get_last_telemetry_record()

```
std::vector< Frame > handle_get_last_telemetry_record (
            const std::string & param,
            OperationType operationType)
```

Handles the get last record command.

This function reads the last record from the telemetry buffer, base64 encodes it, and sends the encoded data as a response.

Parameters

param	Unused.
operationType	The operation type (must be GET).

Returns

A vector of Frames indicating the result of the operation.

- Success: Frame with base64 encoded telemetry data.
- Error: Frame with error message (e.g., "No telemetry data available").

Note

KBST;0;GET;8;2;;TSBK

This command retrieves the last telemetry record from the buffer and sends it base64 encoded.

Command Command ID: 8.2

Definition at line 32 of file telemetry_commands.cpp.

6.9.2.2 handle_get_last_sensor_record()

```
std::vector< Frame > handle_get_last_sensor_record (
            const std::string & param,
            OperationType operationType)
```

Handles the get last sensor record command.

This function retrieves the last sensor record from the telemetry manager, and sends the data as a response.

Parameters

param	Unused.
operationType	The operation type (must be GET).

Returns

A vector of Frames indicating the result of the operation.

Definition at line 65 of file telemetry_commands.cpp.

6.10 Frame Handling

Functions for encoding, decoding and building communication frames.

Functions

- std::string frame_encode (const Frame &frame)

    Encodes a Frame instance into a string.
- Frame frame_decode (const std::string &data)

    Decodes a string into a Frame instance.
- void frame_process (const std::string &data, Interface interface)

    Executes a command based on the command key and the parameter.
- Frame frame_build (OperationType operation, uint8_t group, uint8_t command, const std::string &value, const ValueUnit unitType)

    Builds a Frame instance based on the execution result, group, command, value, and unit.

6.10.1 Detailed Description

Functions for encoding, decoding and building communication frames.

6.10.2 Function Documentation

6.10.2.1 frame_encode()

```
std::string frame_encode (
            const Frame & frame)
```

Encodes a Frame instance into a string.

Parameters

frame	The Frame instance to encode.

Returns

The Frame encoded as a string.

The encoded string includes the frame direction, operation type, group, command, value, and unit, all delimited by the DELIMITER character. The string is encapsulated by FRAME_BEGIN and FRAME_END.

```
Frame myFrame;
myFrame.header = FRAME_BEGIN;
myFrame.direction = 0;
myFrame.operationType = OperationType::GET;
myFrame.group = 1;
myFrame.command = 1;
myFrame.value = "";
myFrame.unit = "";
myFrame.footer = FRAME_END;

std::string encoded = frame_encode(myFrame);
// encoded will be "KBST;0;GET;1;1;;TSBK"
```

Definition at line 37 of file frame.cpp.

6.10.2.2 frame_decode()

```
Frame frame_decode (
            const std::string & data)
```

Decodes a string into a Frame instance.

Parameters

encodedFrame	The string to decode.

Returns

The Frame instance decoded from the string.

Exceptions

std::runtime_error	if the frame is invalid.

The decoded string is expected to be in the format: FRAME_BEGIN;direction;operationType;group;command;value;unit;FRAME↩_END

Definition at line 62 of file frame.cpp.

6.10.2.3 frame_process()

```
void frame_process (
            const std::string & data,
            Interface interface)
```

Executes a command based on the command key and the parameter.

Parameters

data	The Frame data in string format.

Decodes the frame data, extracts the command key, and executes the corresponding command. Sends the response frame. If an error occurs, an error frame is built and sent.

Definition at line 117 of file frame.cpp.

6.10.2.4 frame_build()

```
Frame frame_build (
            OperationType operation,
            uint8_t group,
            uint8_t command,
            const std::string & value,
            const ValueUnit unitType)
```

Builds a Frame instance based on the execution result, group, command, value, and unit.

Parameters

result	The execution result.
group	The group ID.
command	The command ID within the group.
value	The payload value.
unit	The unit of measurement for the payload value.

Returns

The Frame instance.

Definition at line 158 of file frame.cpp.

6.11 Protocol

Definitions for the communication protocol used by the satellite.

Classes

- struct Frame

    Represents a communication frame used for data exchange.

Enumerations

- enum class ErrorCode {
  ErrorCode::PARAM_UNNECESSARY , ErrorCode::PARAM_REQUIRED , ErrorCode::PARAM_INVALID , ErrorCode::INVALID_OPERATION ,
  ErrorCode::NOT_ALLOWED , ErrorCode::INVALID_FORMAT , ErrorCode::INVALID_VALUE , ErrorCode::FAIL_TO_SET
  ,
  ErrorCode::INTERNAL_FAIL_TO_READ , ErrorCode::UNKNOWN_ERROR }

    Standard error codes for command responses.
- enum class OperationType {
  OperationType::GET , OperationType::SET , OperationType::RES , OperationType::VAL ,
  OperationType::SEQ , OperationType::ERR }

    Represents the type of operation being performed.
- enum class CommandAccessLevel { CommandAccessLevel::NONE , CommandAccessLevel::READ_ONLY
  , CommandAccessLevel::WRITE_ONLY , CommandAccessLevel::READ_WRITE }

    Represents the access level required to execute a command.
- enum class ValueUnit {
  ValueUnit::UNDEFINED , ValueUnit::SECOND , ValueUnit::VOLT , ValueUnit::BOOL ,
  ValueUnit::DATETIME , ValueUnit::TEXT , ValueUnit::MILIAMP }

    Represents the unit of measurement for a payload value.
- enum class ExceptionType {
  ExceptionType::NONE , ExceptionType::NOT_ALLOWED , ExceptionType::INVALID_PARAM , ExceptionType::INVALID_OPER
  ,
  ExceptionType::PARAM_UNECESSARY }

    Represents the type of exception that occurred during command execution.
- enum class Interface { Interface::UART , Interface::LORA }

    Represents the communication interface being used.

6.11.1 Detailed Description

Definitions for the communication protocol used by the satellite.

6.11.2 Enumeration Type Documentation

6.11.2.1 ErrorCode

```
enum class ErrorCode  [strong]
```

Standard error codes for command responses.

Enumerator

PARAM_UNNECESSARY	
PARAM_REQUIRED	
PARAM_INVALID	
INVALID_OPERATION	
NOT_ALLOWED	
INVALID_FORMAT	
INVALID_VALUE	
FAIL_TO_SET	
INTERNAL_FAIL_TO_READ	
UNKNOWN_ERROR	

Definition at line 53 of file protocol.h.

6.11.2.2 OperationType

`enum class OperationType [strong]`

Represents the type of operation being performed.

Enumerator

GET	Get data.
SET	Set data.
RES	Set command result.
VAL	Get command value.
SEQ	Sequence element response.
ERR	Error occurred during command execution.

Definition at line 72 of file protocol.h.

6.11.2.3 CommandAccessLevel

`enum class CommandAccessLevel [strong]`

Represents the access level required to execute a command.

Enumerator

NONE	No access allowed.
READ_ONLY	Read-only access.
WRITE_ONLY	Write-only access.
READ_WRITE	Read and write access.

Definition at line 95 of file protocol.h.

6.11.2.4 ValueUnit

`enum class ValueUnit [strong]`

Represents the unit of measurement for a payload value.

Enumerator

UNDEFINED	Unit is undefined.
SECOND	Unit is seconds.
VOLT	Unit is volts.
BOOL	Unit is boolean.
DATETIME	Unit is date and time.
TEXT	Unit is text.
MILIAMP	Unit is milliamperes.

Definition at line 113 of file protocol.h.

6.11.2.5 ExceptionType

```
enum class ExceptionType [strong]
```

Represents the type of exception that occurred during command execution.

Enumerator

NONE	No exception.
NOT_ALLOWED	Operation not allowed.
INVALID_PARAM	Invalid parameter provided.
INVALID_OPERATION	Invalid operation requested.
PARAM_UNECESSARY	Parameter is unnecessary for the operation.

Definition at line 137 of file protocol.h.

6.11.2.6 Interface

```
enum class Interface [strong]
```

Represents the communication interface being used.

Enumerator

UART	UART interface.
LORA	LoRa interface.

Definition at line 157 of file protocol.h.

6.12 Receiving Data

Functions for receiving and processing data from LoRa and UART interfaces.

Functions

- void on_receive (int packet_size)

    Callback function for handling received LoRa packets.
- void handle_uart_input ()

    Handles UART input.

6.12.1 Detailed Description

Functions for receiving and processing data from LoRa and UART interfaces.

6.12.2 Function Documentation

6.12.2.1 on_receive()

```
void on_receive (
            int packet_size)
```

Callback function for handling received LoRa packets.

Parameters

packet_size	The size of the received packet.

Reads the received LoRa packet, extracts metadata, validates the lora_address_remote and local addresses, extracts the frame data, and processes it. Prints raw hex values for debugging.

Definition at line 19 of file receive.cpp.

6.12.2.2 handle_uart_input()

```
void handle_uart_input ()
```

Handles UART input.

Reads characters from the UART port, appends them to a buffer, and processes the buffer when a newline character is received.

Definition at line 90 of file receive.cpp.

6.13 Utility Converters

Functions

- std::string exception_type_to_string (ExceptionType type)

  Converts an ExceptionType to a string.
- std::string value_unit_type_to_string (ValueUnit unit)

  Converts a ValueUnit to a string.
- std::string operation_type_to_string (OperationType type)

  Converts an OperationType to a string.
- OperationType string_to_operation_type (const std::string &str)

  Converts a string to an OperationType.
- std::string error_code_to_string (ErrorCode code)

  Converts an ErrorCode to its string representation.
- std::vector< uint8_t > hex_string_to_bytes (const std::string &hexString)

  Converts a hex string to a vector of bytes.

6.13.1 Detailed Description

6.13.2 Function Documentation

6.13.2.1 exception_type_to_string()

```
std::string exception_type_to_string (
            ExceptionType type)
```

Converts an ExceptionType to a string.

Parameters

type	The ExceptionType to convert.

Returns

The string representation of the ExceptionType.

Definition at line 16 of file utils_converters.cpp.

6.13.2.2 value_unit_type_to_string()

```
std::string value_unit_type_to_string (
            ValueUnit unit)
```

Converts a ValueUnit to a string.

Parameters

unit	The ValueUnit to convert.

Returns

The string representation of the ValueUnit.

Definition at line 34 of file utils_converters.cpp.

6.13.2.3 operation_type_to_string()

```
std::string operation_type_to_string (
            OperationType type)
```

Converts an OperationType to a string.

Parameters

type	The OperationType to convert.

Returns

The string representation of the OperationType.

Definition at line 54 of file utils_converters.cpp.

6.13.2.4 string_to_operation_type()

```
OperationType string_to_operation_type (
            const std::string & str)
```

Converts a string to an OperationType.

Parameters

str	The string to convert.

Returns

The OperationType corresponding to the string. Defaults to GET if the string is not recognized.

Definition at line 73 of file utils_converters.cpp.

6.13.2.5 error_code_to_string()

```
std::string error_code_to_string (
            ErrorCode code)
```

Converts an ErrorCode to its string representation.

Parameters

code	The error code

Returns

String representation of the error code

Definition at line 89 of file utils_converters.cpp.

6.13.2.6 hex_string_to_bytes()

```
std::vector< uint8_t > hex_string_to_bytes (
            const std::string & hexString)
```

Converts a hex string to a vector of bytes.

Parameters

hexString	The hex string to convert.

Returns

A vector of bytes representing the hex string.

Definition at line 111 of file utils_converters.cpp.

6.14 RTC clock

Functions for interfacing with the DS3231 RTC module.

Functions

- DS3231::DS3231 ()

  Constructor for the DS3231 class.
- static DS3231 & DS3231::get_instance ()

  Gets the singleton instance of the DS3231 class.
- int DS3231::set_time (ds3231_data_t ∗data)

  Sets the time on the DS3231 clock.
- int DS3231::get_time (ds3231_data_t ∗data)

  Gets the current time from the DS3231 clock.
- int DS3231::read_temperature (float ∗resolution)

  Reads the current temperature from the DS3231.
- int DS3231::set_unix_time (time_t unix_time)

  Sets the time using a Unix timestamp.
- time_t DS3231::get_unix_time ()

  Gets the current time as a Unix timestamp.
- int DS3231::clock_enable ()

  Enables the DS3231 clock oscillator.
- int16_t DS3231::get_timezone_offset () const

  Gets the current timezone offset.
- void DS3231::set_timezone_offset (int16_t offset_minutes)

  Sets the timezone offset.
- uint32_t DS3231::get_clock_sync_interval () const

  Gets the clock synchronization interval.
- void DS3231::set_clock_sync_interval (uint32_t interval_minutes)

  Sets the clock synchronization interval.
- time_t DS3231::get_last_sync_time () const

  Gets the timestamp of the last clock synchronization.
- void DS3231::update_last_sync_time ()

  Updates the last sync time to current time.
- time_t DS3231::get_local_time ()

  Gets the current local time (including timezone offset)
- bool DS3231::is_sync_needed ()

  Checks if clock synchronization is needed.

- bool DS3231::sync_clock_with_gps ()
  Synchronizes clock with GPS data.
- int DS3231::i2c_read_reg (uint8_t reg_addr, size_t length, uint8_t *data)
  Reads data from a specific register on the DS3231.
- int DS3231::i2c_write_reg (uint8_t reg_addr, size_t length, uint8_t *data)
  Writes data to a specific register on the DS3231.
- uint8_t DS3231::bin_to_bcd (const uint8_t data)
  Converts binary value to BCD (Binary Coded Decimal)
- uint8_t DS3231::bcd_to_bin (const uint8_t bcd)
  Converts BCD (Binary Coded Decimal) to binary value.

6.14.1 Detailed Description

Functions for interfacing with the DS3231 RTC module.

6.14.2 Function Documentation

6.14.2.1 DS3231()

```
DS3231::DS3231 ()  [private]
```

Constructor for the DS3231 class.

Initializes the I2C interface and sets the device address for the DS3231 RTC module. The constructor is private to enforce the singleton pattern, ensuring that only one instance of the class can be created. The mutex for the class is also initialized.

Note

The DS3231 device address is defined in the header file as DS3231_DEVICE_ADRESS.

Definition at line 23 of file DS3231.cpp.

6.14.2.2 get_instance()

```
DS3231 & DS3231::get_instance ()  [static]
```

Gets the singleton instance of the DS3231 class.

Returns

A reference to the singleton instance of the DS3231 class

A reference to the singleton instance of the DS3231 class

This function provides access to the single instance of the DS3231 class, ensuring that only one object manages the RTC module. The instance is created upon the first call to this function and remains available for the lifetime of the program.

Definition at line 38 of file DS3231.cpp.

6.14.2.3 set_time()

```
int DS3231::set_time (
            ds3231_data_t * data)
```

Sets the time on the DS3231 clock.

Sets the time on the DS3231 RTC module.

Parameters

in	*data*	Pointer to a ds3231_data_t structure with time information

Returns

0 on success, -1 on failure

Parameters

in	*data*	Pointer to a ds3231_data_t structure containing the time to set

Returns

0 on success, -1 on failure

Writes time and date data to the DS3231 module. The function performs input validation to ensure that the provided values are within valid ranges. The time values are converted from binary to BCD format before being written to the device registers.

Note

The ds3231_data_t structure must contain valid values for seconds, minutes, hours, day, date, month, year, and century.

Definition at line 56 of file DS3231.cpp.

6.14.2.4 get_time()

```
int DS3231::get_time (
            ds3231_data_t * data)
```

Gets the current time from the DS3231 clock.

Reads the current time from the DS3231 RTC module.

Parameters

out	*data*	Pointer to a ds3231_data_t structure to store time information

Returns

0 on success, -1 on failure

Parameters

out	*data*	Pointer to a ds3231_data_t structure to store the read time

Returns

0 on success, -1 on failure

Reads the time and date registers from the DS3231 and stores the decoded values in the provided data structure. The BCD values from the registers are converted to binary format. The function performs validation on the read values to ensure they are within valid ranges.

Note

The function logs debug information including the raw BCD values read and the decoded time and date.

Definition at line 126 of file DS3231.cpp.

6.14.2.5 read_temperature()

```
int DS3231::read_temperature (
            float * resolution)
```

Reads the current temperature from the DS3231.

Reads the temperature from the DS3231's internal temperature sensor.

Parameters

out	*resolution*	Pointer to store the temperature value in Celsius

Returns

0 on success, -1 on failure

Parameters

out	*resolution*	Pointer to a float to store the temperature value

Returns

0 on success, -1 on failure

The DS3231 includes an internal temperature sensor with 0.25℃ resolution. This function reads the sensor value and calculates the temperature in degrees Celsius. The temperature sensor is primarily used for the oscillator's temperature compensation, but can be used for general temperature monitoring as well.

Definition at line 168 of file DS3231.cpp.

6.14.2.6 set_unix_time()

```
int DS3231::set_unix_time (
            time_t unix_time)
```

Sets the time using a Unix timestamp.

Sets the DS3231 clock using a Unix timestamp.

Parameters

in	*unix_time*	Time as seconds since Unix epoch (1970-01-01 00:00:00 UTC)

Returns

0 on success, -1 on failure

Parameters

in	*unix_time*	The time in seconds since the Unix epoch (1970-01-01 00:00:00 UTC)

Returns

0 on success, -1 on failure

Converts the provided Unix timestamp to a calendar date and time and sets the DS3231 RTC accordingly. This function properly handles the conversion between the tm structure (used by C standard library) and the internal ds3231_data_t format.

Definition at line 198 of file DS3231.cpp.

6.14.2.7 get_unix_time()

```
time_t DS3231::get_unix_time ()
```

Gets the current time as a Unix timestamp.

Gets the current time from DS3231 as a Unix timestamp.

Returns

Unix timestamp, or -1 on error

Unix timestamp (seconds since 1970-01-01 00:00:00 UTC), or -1 on error

Reads the current time from the DS3231 RTC and converts it to a Unix timestamp. This function properly handles the conversion between the internal ds3231_data_t format and the tm structure used by the C standard library.

Definition at line 229 of file DS3231.cpp.

6.14.2.8 clock_enable()

```
int DS3231::clock_enable ()
```

Enables the DS3231 clock oscillator.

Enables the DS3231's oscillator.

Returns

0 on success, -1 on failure

0 on success, -1 on failure

Reads the control register and clears the EOSC (Enable Oscillator) bit to ensure the oscillator is running. This is necessary for the RTC to keep time when not on external power.

Definition at line 267 of file DS3231.cpp.

6.14.2.9 get_timezone_offset()

```
int16_t DS3231::get_timezone_offset () const
```

Gets the current timezone offset.

Gets the currently configured timezone offset.

Returns

Timezone offset in minutes (-720 to +720)

The timezone offset in minutes

Returns the current timezone offset in minutes relative to UTC. Positive values represent timezones ahead of UTC (east), negative values represent timezones behind UTC (west).

Definition at line 300 of file DS3231.cpp.

6.14.2.10 set_timezone_offset()

```
void DS3231::set_timezone_offset (
             int16_t offset_minutes)
```

Sets the timezone offset.

Parameters

	offset_minutes	Offset in minutes (-720 to +720)
in	*offset_minutes*	The timezone offset in minutes

Sets the timezone offset in minutes relative to UTC. This value is used when converting between UTC and local time. The function validates that the offset is within a valid range (-720 to +720 minutes, which corresponds to -12 to +12 hours).

Note

This setting is stored in memory and does not persist across reboots.

Definition at line 316 of file DS3231.cpp.

6.14.2.11 get_clock_sync_interval()

```
uint32_t DS3231::get_clock_sync_interval () const
```

Gets the clock synchronization interval.

Gets the currently configured clock synchronization interval.

Returns

Sync interval in minutes

The sync interval in minutes

Returns the current interval between clock synchronization attempts. This is the time after which is_sync_needed() will return true.

Definition at line 334 of file DS3231.cpp.

6.14.2.12 set_clock_sync_interval()

```
void DS3231::set_clock_sync_interval (
             uint32_t interval_minutes)
```

Sets the clock synchronization interval.

Parameters

	interval_minutes	Interval in minutes (1-43200)
`in`	*interval_minutes*	The desired sync interval in minutes

Sets how frequently the clock should be synchronized with an external time source (such as GPS). The function validates that the interval is within a valid range (1 minute to 43200 minutes, which is 30 days).

Note

This setting is stored in memory and does not persist across reboots.

Definition at line 349 of file DS3231.cpp.

6.14.2.13 get_last_sync_time()

```
time_t DS3231::get_last_sync_time () const
```

Gets the timestamp of the last clock synchronization.

Gets the timestamp of the last successful clock synchronization.

Returns

Unix timestamp of last sync, 0 if never synced

Unix timestamp of the last sync, or 0 if never synced

Returns the Unix timestamp of when the clock was last successfully synchronized with an external time source. A value of 0 indicates that the clock has never been synchronized.

Definition at line 367 of file DS3231.cpp.

6.14.2.14 update_last_sync_time()

```
void DS3231::update_last_sync_time ()
```

Updates the last sync time to current time.

Updates the last sync timestamp to the current time.

Records the current time as the last successful synchronization time. This should be called after successfully setting the time from an external source (such as GPS). The function logs the update with an informational message.

Definition at line 381 of file DS3231.cpp.

6.14.2.15 get_local_time()

```
time_t DS3231::get_local_time ()
```

Gets the current local time (including timezone offset)

Gets the current local time by applying the timezone offset to UTC time.

Returns

Unix timestamp adjusted for timezone, or -1 on error

Local time as Unix timestamp, or -1 on error

Retrieves the current UTC time from the RTC and applies the configured timezone offset (in minutes) to calculate the local time.

Definition at line 395 of file DS3231.cpp.

6.14.2.16 is_sync_needed()

```
bool DS3231::is_sync_needed ()
```

Checks if clock synchronization is needed.

Determines if the clock needs synchronization based on the configured interval.

Returns

true if sync interval has elapsed since last sync, false otherwise

true if synchronization is needed, false otherwise

This method checks if the clock has ever been synchronized (last_sync_time_ is 0) or if the time elapsed since the last synchronization exceeds the configured sync_interval_minutes_. If the current time cannot be determined, it assumes synchronization is needed.

Definition at line 415 of file DS3231.cpp.

6.14.2.17 sync_clock_with_gps()

```
bool DS3231::sync_clock_with_gps ()
```

Synchronizes clock with GPS data.

Synchronizes the RTC with time from GPS.

Returns

true if sync successful, false otherwise

Parameters

in	*nmea_data*	Reference to NMEA data containing time information

Returns

true if synchronization succeeded, false if it failed

This method attempts to extract valid time data from the provided NMEA data and use it to update the RTC. It performs validity checks on the GPS data before attempting synchronization. If successful, it updates the last sync time and emits a SYNCED event. If unsuccessful, it emits a SYNC_FAILED event.

Note

This function emits events to the EventEmitter system that can be monitored by other components of the system.

Definition at line 446 of file DS3231.cpp.

6.14.2.18 i2c_read_reg()

```
int DS3231::i2c_read_reg (
            uint8_t reg_addr,
            size_t length,
            uint8_t * data)  [private]
```

Reads data from a specific register on the DS3231.

Parameters

in	*reg_addr*	Register address to read from
in	*length*	Number of bytes to read
out	*data*	Buffer to store read data

Returns

0 on success, -1 on failure

Parameters

in	*reg_addr*	Register address to read from
in	*length*	Number of bytes to read
out	*data*	Buffer to store read data

Returns

0 on success, -1 on failure

This method performs a thread-safe I^2C read operation from the DS3231. It first writes the register address to the device, then reads the requested number of bytes. All access is protected by a mutex to prevent concurrent I^2C operations that could corrupt data.

Note

This is a low-level method used internally by the class.

Definition at line 493 of file DS3231.cpp.

6.14.2.19 i2c_write_reg()

```
int DS3231::i2c_write_reg (
            uint8_t reg_addr,
            size_t length,
            uint8_t * data) [private]
```

Writes data to a specific register on the DS3231.

Parameters

in	*reg_addr*	Register address to write to
in	*length*	Number of bytes to write
in	*data*	Buffer containing data to write

Returns

0 on success, -1 on failure

Parameters

in	*reg_addr*	Register address to write to
in	*length*	Number of bytes to write
in	*data*	Buffer containing data to write

Returns

0 on success, -1 on failure

This method performs a thread-safe I^2C write operation to the DS3231. It combines the register address and data into a single buffer and sends it to the device. All access is protected by a mutex to prevent concurrent I^2C operations that could corrupt data.

Note

This is a low-level method used internally by the class.

Definition at line 535 of file DS3231.cpp.

6.14.2.20 bin_to_bcd()

```
uint8_t DS3231::bin_to_bcd (
            const uint8_t data) [private]
```

Converts binary value to BCD (Binary Coded Decimal)

Converts binary value to Binary-Coded Decimal (BCD) format.

Parameters

in	*data*	Binary value to convert (0-99)

Returns

BCD representation of the input value

Parameters

in	*data*	Binary value to convert (0-99)

Returns

BCD representation of the input value

The DS3231 stores time values in BCD format where each nibble represents a decimal digit. This function converts a standard binary value to its BCD equivalent (e.g., 42 becomes 0x42).

Definition at line 566 of file DS3231.cpp.

6.14.2.21 bcd_to_bin()

```
uint8_t DS3231::bcd_to_bin (
            const uint8_t bcd)  [private]
```

Converts BCD (Binary Coded Decimal) to binary value.

Converts Binary-Coded Decimal (BCD) to binary value.

Parameters

in	*bcd*	BCD value to convert

Returns

Binary representation of the input BCD value

Parameters

in	*bcd*	BCD value to convert

Returns

Binary representation of the input BCD value

The DS3231 stores time values in BCD format where each nibble represents a decimal digit. This function converts a BCD value to its standard binary equivalent (e.g., 0x42 becomes 42).

Definition at line 583 of file DS3231.cpp.

6.15 Event Management

Classes and enums for handling system events.

Classes

- class EventLog

    Structure for storing event log data.
- class EventManager

    Manages event logging and storage.
- class EventEmitter

    Provides a simple interface for emitting events.

Macros

- #define EVENT_BUFFER_SIZE 100

    Size of the event buffer.
- #define EVENT_FLUSH_THRESHOLD 10

    Number of events to accumulate before flushing to storage.
- #define EVENT_LOG_FILE "/event_log.csv"

    Path to the event log file.

Enumerations

- enum class EventGroup : uint8_t {
  EventGroup::SYSTEM = 0x00 , EventGroup::POWER = 0x01 , EventGroup::COMMS = 0x02 ,
  EventGroup::GPS = 0x03 ,
  EventGroup::CLOCK = 0x04 }

    Enumeration of event groups.
- enum class SystemEvent : uint8_t {
  SystemEvent::BOOT = 0x01 , SystemEvent::SHUTDOWN = 0x02 , SystemEvent::WATCHDOG_RESET =
  0x03 , SystemEvent::CORE1_START = 0x04 ,
  SystemEvent::CORE1_STOP = 0x05 }

    Enumeration of system events.
- enum class PowerEvent : uint8_t {
  PowerEvent::BATTERY_LOW = 0x01 , PowerEvent::BATTERY_FULL = 0x02 , PowerEvent::POWER_FALLING
  = 0x03 , PowerEvent::BATTERY_NORMAL = 0x04 ,
  PowerEvent::SOLAR_ACTIVE = 0x05 , PowerEvent::SOLAR_INACTIVE = 0x06 , PowerEvent::USB_CONNECTED
  = 0x07 , PowerEvent::USB_DISCONNECTED = 0x08 }

    Enumeration of power events.
- enum class CommsEvent : uint8_t {
  CommsEvent::RADIO_INIT = 0x01 , CommsEvent::RADIO_ERROR = 0x02 , CommsEvent::MSG_RECEIVED
  = 0x03 , CommsEvent::MSG_SENT = 0x04 ,
  CommsEvent::UART_ERROR = 0x06 }

    Enumeration of communications events.
- enum class GPSEvent : uint8_t {
  GPSEvent::LOCK = 0x01 , GPSEvent::LOST = 0x02 , GPSEvent::ERROR = 0x03 , GPSEvent::POWER_ON
  = 0x04 ,
  GPSEvent::POWER_OFF = 0x05 , GPSEvent::DATA_READY = 0x06 , GPSEvent::PASS_THROUGH_START
  = 0x07 , GPSEvent::PASS_THROUGH_END = 0x08 }

    Enumeration of GPS events.
- enum class ClockEvent : uint8_t { ClockEvent::CHANGED = 0x01 , ClockEvent::GPS_SYNC = 0x02 ,
  ClockEvent::GPS_SYNC_DATA_NOT_READY = 0x03 }

    Enumeration of clock events.

Functions

- class EventLog __attribute__ ((packed))
- bool EventManager::init ()

  Initializes the event manager.
- void EventManager::log_event (uint8_t group, uint8_t event)

  Logs an event to the event buffer.
- const EventLog & EventManager::get_event (size_t index) const

  Gets an event from the event buffer.
- bool EventManager::save_to_storage ()

  Saves the event buffer to persistent storage.

Variables

- class EventManager __attribute__

6.15.1 Detailed Description

Classes and enums for handling system events.

6.15.2 Macro Definition Documentation

6.15.2.1 EVENT_BUFFER_SIZE

```
#define EVENT_BUFFER_SIZE 100
```

Size of the event buffer.

Definition at line 32 of file event_manager.h.

6.15.2.2 EVENT_FLUSH_THRESHOLD

```
#define EVENT_FLUSH_THRESHOLD 10
```

Number of events to accumulate before flushing to storage.

Definition at line 37 of file event_manager.h.

6.15.2.3 EVENT_LOG_FILE

```
#define EVENT_LOG_FILE "/event_log.csv"
```

Path to the event log file.

Definition at line 42 of file event_manager.h.

6.15.3 Enumeration Type Documentation

6.15.3.1 EventGroup

```
enum class EventGroup : uint8_t [strong]
```

Enumeration of event groups.

Defines the different categories of events that can be logged.

Enumerator

SYSTEM	System-level events.
POWER	Power management events.
COMMS	Communications events.
GPS	GPS events.
CLOCK	Clock events.

Definition at line 50 of file event_manager.h.

6.15.3.2 SystemEvent

```
enum class SystemEvent : uint8_t [strong]
```

Enumeration of system events.

Defines specific system-level events.

Enumerator

BOOT	System boot event.
SHUTDOWN	System shutdown event.
WATCHDOG_RESET	Watchdog reset event.
CORE1_START	Core 1 start event.
CORE1_STOP	Core 1 stop event.

Definition at line 69 of file event_manager.h.

6.15.3.3 PowerEvent

```
enum class PowerEvent : uint8_t [strong]
```

Enumeration of power events.

Defines specific power management events.

Enumerator

BATTERY_LOW	Low battery event.
BATTERY_FULL	Overcharge event.
POWER_FALLING	Power falling event.
BATTERY_NORMAL	Power normal event.
SOLAR_ACTIVE	Solar charging active event.
SOLAR_INACTIVE	Solar charging inactive event.
USB_CONNECTED	USB connected event.
USB_DISCONNECTED	USB disconnected event.

Definition at line 87 of file event_manager.h.

6.15.3.4 CommsEvent

```
enum class CommsEvent : uint8_t [strong]
```

Enumeration of communications events.

Defines specific communications events.

Enumerator

RADIO_INIT	Radio initialization event.
RADIO_ERROR	Radio error event.
MSG_RECEIVED	Message received event.
MSG_SENT	Message sent event.
UART_ERROR	UART error event.

Definition at line 112 of file event_manager.h.

6.15.3.5 GPSEvent

```
enum class GPSEvent : uint8_t [strong]
```

Enumeration of GPS events.

Defines specific GPS events.

Enumerator

LOCK	GPS lock event.
LOST	GPS lost event.
ERROR	GPS error event.
POWER_ON	GPS power on event.
POWER_OFF	GPS power off event.
DATA_READY	GPS data ready event.
PASS_THROUGH_START	GPS pass-through start event.
PASS_THROUGH_END	GPS pass-through end event.

Definition at line 130 of file event_manager.h.

6.15.3.6 ClockEvent

```
enum class ClockEvent : uint8_t [strong]
```

Enumeration of clock events.

Defines specific clock-related events.

Enumerator

CHANGED	Clock changed event.
GPS_SYNC	GPS sync event.
GPS_SYNC_DATA_NOT_READY	GPS sync data not ready event.

Definition at line 154 of file event_manager.h.

6.15.4 Function Documentation

6.15.4.1 __attribute__()

```
class EventLog __attribute__ (
            (packed) )
```

6.15.4.2 init()

```
bool EventManager::init ()
```

Initializes the event manager.

Returns

True if initialization was successful, false otherwise.

Definition at line 30 of file event_manager.cpp.

6.15.4.3 log_event()

```
void EventManager::log_event (
            uint8_t group,
            uint8_t event)
```

Logs an event to the event buffer.

Parameters

in	*group*	Event group.
in	*event*	Event code.

Definition at line 63 of file event_manager.cpp.

6.15.4.4 get_event()

```
const EventLog & EventManager::get_event (
            size_t index) const
```

Gets an event from the event buffer.

Parameters

in	*index*	Index of the event to retrieve.

Returns

A const reference to the event log entry.

Definition at line 102 of file event_manager.cpp.

6.15.4.5 save_to_storage()

```
bool EventManager::save_to_storage ()
```

Saves the event buffer to persistent storage.

Returns

True if the save was successful, false otherwise.

Definition at line 128 of file event_manager.cpp.

6.15.5 Variable Documentation

6.15.5.1 __attribute__

```
class EventManager __attribute__
```

6.16 Location

Classes for handling location data.

Classes

- class NMEAData

  Manages parsed NMEA sentences.

Macros

- #define MAX_RAW_DATA_LENGTH 256

  Maximum length of the raw data buffer for NMEA sentences.

Functions

- std::vector< std::string > splitString (const std::string &str, char delimiter)

  Splits a string into tokens based on a delimiter.
- void collect_gps_data ()

  Collects GPS data from the UART and updates the NMEA data.

6.16.1 Detailed Description

Classes for handling location data.

6.16.2 Macro Definition Documentation

6.16.2.1 MAX_RAW_DATA_LENGTH

```
#define MAX_RAW_DATA_LENGTH 256
```

Maximum length of the raw data buffer for NMEA sentences.

Definition at line 30 of file gps_collector.cpp.

6.16.3 Function Documentation

6.16.3.1 splitString()

```
std::vector< std::string > splitString (
            const std::string & str,
            char delimiter)
```

Splits a string into tokens based on a delimiter.

Parameters

in	*str*	The string to split.
in	*delimiter*	The delimiter character.

Returns

A vector of strings representing the tokens.

Definition at line 40 of file gps_collector.cpp.

6.16.3.2 collect_gps_data()

```
void collect_gps_data ()
```

Collects GPS data from the UART and updates the NMEA data.

This function reads raw NMEA sentences from the GPS UART, parses them, and updates the RMC and GGA tokens in the NMEAData singleton. It also handles buffer overflow and checks for bootloader reset pending status.

Definition at line 59 of file gps_collector.cpp.

6.17 INA3221 Power Monitor

Topics

- Configuration Functions
- Measurement Functions

6.17.1 Detailed Description

6.17.2 Configuration Functions

Functions

- INA3221::INA3221 (ina3221_addr_t addr, i2c_inst_t *i2c)

  Constructor for INA3221 class.
- bool INA3221::begin ()

  Initialize the INA3221 device.
- void INA3221::reset ()

  Reset the INA3221 to default settings.
- uint16_t INA3221::get_manufacturer_id ()

  Get the manufacturer ID of the device.
- uint16_t INA3221::get_die_id ()

  Get the die ID of the device.
- uint16_t INA3221::read_register (ina3221_reg_t reg)

  Read a register from the device.
- void INA3221::set_mode_power_down ()

  Set device to power-down mode.
- void INA3221::set_mode_continuous ()

  Set device to continuous measurement mode.
- void INA3221::set_mode_triggered ()

  Set device to triggered measurement mode.
- void INA3221::set_shunt_measurement_enable ()

  Enable shunt voltage measurements.
- void INA3221::set_shunt_measurement_disable ()

  Disable shunt voltage measurements.
- void INA3221::set_bus_measurement_enable ()

  Enable bus voltage measurements.
- void INA3221::set_bus_measurement_disable ()

  Disable bus voltage measurements.
- void INA3221::set_averaging_mode (ina3221_avg_mode_t mode)

  Set the averaging mode for measurements.
- void INA3221::set_bus_conversion_time (ina3221_conv_time_t convTime)

  Set bus voltage conversion time.
- void INA3221::set_shunt_conversion_time (ina3221_conv_time_t convTime)

  Set shunt voltage conversion time.

6.17.2.1 Detailed Description

Functions for configuring the INA3221 device

6.17.2.2 Function Documentation

6.17.2.2.1 INA3221()

```
INA3221::INA3221 (
            ina3221_addr_t addr,
            i2c_inst_t * i2c)
```

Constructor for INA3221 class.

Parameters

addr	I2C address of the device
i2c	Pointer to I2C instance

Definition at line 41 of file INA3221.cpp.

6.17.2.2.2 begin()

```
bool INA3221::begin ()
```

Initialize the INA3221 device.

Returns

true if initialization successful, false otherwise

Sets up shunt resistors, filter resistors, and verifies device IDs

Definition at line 51 of file INA3221.cpp.

6.17.2.2.3 reset()

```
void INA3221::reset ()
```

Reset the INA3221 to default settings.

Performs a software reset of the device by setting the reset bit

Definition at line 84 of file INA3221.cpp.

6.17.2.2.4 get_manufacturer_id()

```
uint16_t INA3221::get_manufacturer_id ()
```

Get the manufacturer ID of the device.

Returns

16-bit manufacturer ID (should be 0x5449)

Definition at line 98 of file INA3221.cpp.

6.17.2.2.5 get_die_id()

```
uint16_t INA3221::get_die_id ()
```

Get the die ID of the device.

Returns

16-bit die ID (should be 0x3220)

Definition at line 110 of file INA3221.cpp.

6.17.2.2.6 read_register()

```
uint16_t INA3221::read_register (
              ina3221_reg_t reg)
```

Read a register from the device.

Parameters

reg	Register address to read

Returns

16-bit value read from the register

Definition at line 123 of file INA3221.cpp.

6.17.2.2.7 set_mode_power_down()

```
void INA3221::set_mode_power_down ()
```

Set device to power-down mode.

Disables bus voltage and continuous measurements

Definition at line 137 of file INA3221.cpp.

6.17.2.2.8 set_mode_continuous()

```
void INA3221::set_mode_continuous ()
```

Set device to continuous measurement mode.

Enables continuous measurement of bus voltage and shunt voltage

Definition at line 152 of file INA3221.cpp.

6.17.2.2.9 set_mode_triggered()

```
void INA3221::set_mode_triggered ()
```

Set device to triggered measurement mode.

Disables continuous measurements, requiring manual triggers

Definition at line 166 of file INA3221.cpp.

6.17.2.2.10 set_shunt_measurement_enable()

```
void INA3221::set_shunt_measurement_enable ()
```

Enable shunt voltage measurements.

Definition at line 179 of file INA3221.cpp.

6.17.2.2.11 set_shunt_measurement_disable()

```
void INA3221::set_shunt_measurement_disable ()
```

Disable shunt voltage measurements.

Definition at line 192 of file INA3221.cpp.

6.17.2.2.12 set_bus_measurement_enable()

```
void INA3221::set_bus_measurement_enable ()
```

Enable bus voltage measurements.

Definition at line 205 of file INA3221.cpp.

6.17.2.2.13 set_bus_measurement_disable()

```
void INA3221::set_bus_measurement_disable ()
```

Disable bus voltage measurements.

Definition at line 218 of file INA3221.cpp.

6.17.2.2.14 set_averaging_mode()

```
void INA3221::set_averaging_mode (
             ina3221_avg_mode_t mode)
```

Set the averaging mode for measurements.

Parameters

mode	Number of samples to average

Definition at line 232 of file INA3221.cpp.

6.17.2.2.15 set_bus_conversion_time()

```
void INA3221::set_bus_conversion_time (
             ina3221_conv_time_t convTime)
```

Set bus voltage conversion time.

Parameters

convTime	Conversion time setting

Definition at line 246 of file INA3221.cpp.

6.17.2.2.16 set_shunt_conversion_time()

```
void INA3221::set_shunt_conversion_time (
             ina3221_conv_time_t convTime)
```

Set shunt voltage conversion time.

Parameters

convTime	Conversion time setting

Definition at line 260 of file INA3221.cpp.

6.17.3 Measurement Functions

Functions

- int32_t INA3221::get_shunt_voltage (ina3221_ch_t channel)

  Get shunt voltage for a specific channel.
- float INA3221::get_current_ma (ina3221_ch_t channel)

  Get current for a specific channel.
- float INA3221::get_voltage (ina3221_ch_t channel)

  Get bus voltage for a specific channel.

6.17.3.1 Detailed Description

Functions for reading voltage, current and power measurements

6.17.3.2 Function Documentation

6.17.3.2.1 get_shunt_voltage()

```
int32_t INA3221::get_shunt_voltage (
            ina3221_ch_t channel)
```

Get shunt voltage for a specific channel.

Parameters

channel	Channel number (1-3)

Returns

Shunt voltage in microvolts (µV)

Definition at line 276 of file INA3221.cpp.

6.17.3.2.2 get_current_ma()

```
float INA3221::get_current_ma (
            ina3221_ch_t channel)
```

Get current for a specific channel.

Parameters

channel	Channel number (1-3)

Returns

Current in milliamps (mA)

Definition at line 308 of file INA3221.cpp.

6.17.3.2.3 get_voltage()

```
float INA3221::get_voltage (
            ina3221_ch_t channel)
```

Get bus voltage for a specific channel.

Parameters

channel	Channel number (1-3)

Returns

Voltage in volts (V)

Definition at line 324 of file INA3221.cpp.

6.18 Power Management

Classes for handling power-related functions.

Classes

- class PowerManager

  Manages power-related functions.

Functions

- PowerManager::PowerManager ()
  Private constructor for the singleton pattern.
- static PowerManager & PowerManager::get_instance ()
  Gets the singleton instance of the PowerManager class.
- bool PowerManager::initialize ()
  Initializes the PowerManager.
- std::string PowerManager::read_device_ids ()
  Reads the manufacturer and die IDs from the INA3221.
- float PowerManager::get_voltage_battery ()
  Gets the battery voltage.
- float PowerManager::get_voltage_5v ()
  Gets the 5V voltage.
- float PowerManager::get_current_charge_usb ()
  Gets the USB charging current.
- float PowerManager::get_current_draw ()
  Gets the current draw.
- float PowerManager::get_current_charge_solar ()
  Gets the solar charging current.
- float PowerManager::get_current_charge_total ()
  Gets the total charging current.
- void PowerManager::configure (const std::map< std::string, std::string > &config)
  Configures the INA3221.
- bool PowerManager::is_charging_solar ()
  Checks if solar charging is active.
- bool PowerManager::is_charging_usb ()
  Checks if USB charging is active.

6.18.1 Detailed Description

Classes for handling power-related functions.

6.18.2 Function Documentation

6.18.2.1 PowerManager()

```
PowerManager::PowerManager () [private]
```

Private constructor for the singleton pattern.

Initializes the INA3221 and mutex.

Definition at line 25 of file PowerManager.cpp.

6.18.2.2 get_instance()

```
PowerManager & PowerManager::get_instance ()  [static]
```

Gets the singleton instance of the PowerManager class.

Returns

A reference to the singleton instance.

Definition at line 35 of file PowerManager.cpp.

6.18.2.3 initialize()

```
bool PowerManager::initialize ()
```

Initializes the PowerManager.

Returns

True if initialization was successful, false otherwise.

Definition at line 45 of file PowerManager.cpp.

6.18.2.4 read_device_ids()

```
std::string PowerManager::read_device_ids ()
```

Reads the manufacturer and die IDs from the INA3221.

Returns

A string containing the manufacturer and die IDs.

Definition at line 58 of file PowerManager.cpp.

6.18.2.5 get_voltage_battery()

```
float PowerManager::get_voltage_battery ()
```

Gets the battery voltage.

Returns

The battery voltage in volts.

Definition at line 77 of file PowerManager.cpp.

6.18.2.6 get_voltage_5v()

```
float PowerManager::get_voltage_5v ()
```

Gets the 5V voltage.

Returns

The 5V voltage in volts.

Definition at line 90 of file PowerManager.cpp.

6.18.2.7 get_current_charge_usb()

```
float PowerManager::get_current_charge_usb ()
```

Gets the USB charging current.

Returns

The USB charging current in milliamperes.

Definition at line 103 of file PowerManager.cpp.

6.18.2.8 get_current_draw()

```
float PowerManager::get_current_draw ()
```

Gets the current draw.

Returns

The current draw in milliamperes.

Definition at line 116 of file PowerManager.cpp.

6.18.2.9 get_current_charge_solar()

```
float PowerManager::get_current_charge_solar ()
```

Gets the solar charging current.

Returns

The solar charging current in milliamperes.

Definition at line 129 of file PowerManager.cpp.

6.18.2.10 get_current_charge_total()

```
float PowerManager::get_current_charge_total ()
```

Gets the total charging current.

Returns

The total charging current in milliamperes.

Definition at line 142 of file PowerManager.cpp.

6.18.2.11 configure()

```
void PowerManager::configure (
            const std::map< std::string, std::string > & config)
```

Configures the INA3221.

Parameters

in	*config*	A map of configuration parameters.

Definition at line 155 of file PowerManager.cpp.

6.18.2.12 is_charging_solar()

```
bool PowerManager::is_charging_solar ()
```

Checks if solar charging is active.

Returns

True if solar charging is active, false otherwise.

Definition at line 189 of file PowerManager.cpp.

6.18.2.13 is_charging_usb()

```
bool PowerManager::is_charging_usb ()
```

Checks if USB charging is active.

Returns

True if USB charging is active, false otherwise.

Definition at line 202 of file PowerManager.cpp.

6.19 BH1750 Light Sensor

Driver for the BH1750 digital light sensor.

Topics

- Constants

  Defines constants used by the BH1750 driver.
- Types

  Defines types used by the BH1750 driver.

Classes

- class BH1750

  Class to interface with the BH1750 light sensor.

Functions

- BH1750::BH1750 (i2c_inst_t *i2c, uint8_t addr=0x23)
  Constructor for the BH1750 class.
- bool BH1750::begin (Mode mode=Mode::CONTINUOUS_HIGH_RES_MODE)
  Initializes the BH1750 sensor.
- bool BH1750::configure (Mode mode)
  Configures the BH1750 sensor with the specified mode.
- float BH1750::get_light_level ()
  Reads the light level from the BH1750 sensor.
- void BH1750::write8 (uint8_t data)
  Writes a single byte of data to the BH1750 sensor.

6.19.1 Detailed Description

Driver for the BH1750 digital light sensor.

6.19.2 Function Documentation

6.19.2.1 BH1750()

```
BH1750::BH1750 (
            i2c_inst_t * i2c,
            uint8_t addr = 0x23)
```

Constructor for the BH1750 class.

Parameters

i2c	Pointer to the I2C interface.
addr	I2C address of the BH1750 sensor (default: 0x23).

Definition at line 20 of file BH1750.cpp.

6.19.2.2 begin()

```
bool BH1750::begin (
            Mode mode = Mode::CONTINUOUS_HIGH_RES_MODE)
```

Initializes the BH1750 sensor.

Parameters

mode	Measurement mode to use (default: CONTINUOUS_HIGH_RES_MODE).

Returns

True if initialization was successful, false otherwise.

Definition at line 28 of file BH1750.cpp.

6.19.2.3 configure()

```
bool BH1750::configure (
            Mode mode)
```

Configures the BH1750 sensor with the specified mode.

Parameters

mode	Measurement mode to configure.

Returns

True if configuration was successful, false otherwise.

Definition at line 42 of file BH1750.cpp.

6.19.2.4 get_light_level()

```
float BH1750::get_light_level ()
```

Reads the light level from the BH1750 sensor.

Returns

Light level in lux.

Definition at line 67 of file BH1750.cpp.

6.19.2.5 write8()

```
void BH1750::write8 (
            uint8_t data)  [private]
```

Writes a single byte of data to the BH1750 sensor.

Parameters

data	Byte of data to write.

Definition at line 81 of file BH1750.cpp.

6.19.3 Constants

Defines constants used by the BH1750 driver.

Macros

- #define _BH1750_DEVICE_ID 0xE1

  Correct content of WHO_AM_I register (not actually used in this driver).
- #define _BH1750_MTREG_MIN 31

  Minimum value for the MTREG register.
- #define _BH1750_MTREG_MAX 254

  Maximum value for the MTREG register.
- #define _BH1750_DEFAULT_MTREG 69

  Default value for the MTREG register.

6.19.3.1 Detailed Description

Defines constants used by the BH1750 driver.

6.19.3.2 Macro Definition Documentation

6.19.3.2.1 _BH1750_DEVICE_ID

```
#define _BH1750_DEVICE_ID 0xE1
```

Correct content of WHO_AM_I register (not actually used in this driver).

Definition at line 36 of file BH1750.h.

6.19.3.2.2 _BH1750_MTREG_MIN

```
#define _BH1750_MTREG_MIN 31
```

Minimum value for the MTREG register.

Definition at line 42 of file BH1750.h.

6.19.3.2.3 _BH1750_MTREG_MAX

```
#define _BH1750_MTREG_MAX 254
```

Maximum value for the MTREG register.

Definition at line 48 of file BH1750.h.

6.19.3.2.4 _BH1750_DEFAULT_MTREG

```
#define _BH1750_DEFAULT_MTREG 69
```

Default value for the MTREG register.

Definition at line 54 of file BH1750.h.

6.19.4 Types

Defines types used by the BH1750 driver.

Enumerations

- enum class BH1750::Mode : uint8_t {
BH1750::Mode::UNCONFIGURED_POWER_DOWN = 0x00 , BH1750::Mode::POWER_ON = 0x01 , BH1750::Mode::RESET = 0x07 , BH1750::Mode::CONTINUOUS_HIGH_RES_MODE = 0x10 , BH1750::Mode::CONTINUOUS_HIGH_RES_MODE_2 = 0x11 , BH1750::Mode::CONTINUOUS_LOW_RES_MODE = 0x13 , BH1750::Mode::ONE_TIME_HIGH_RES_MODE = 0x20 , BH1750::Mode::ONE_TIME_HIGH_RES_MODE_2 = 0x21 ,
BH1750::Mode::ONE_TIME_LOW_RES_MODE = 0x23 }

  Enumeration of measurement modes for the BH1750 sensor.

6.19.4.1 Detailed Description

Defines types used by the BH1750 driver.

6.19.4.2 Enumeration Type Documentation

6.19.4.2.1 Mode

```
enum class BH1750::Mode : uint8_t [strong]
```

Enumeration of measurement modes for the BH1750 sensor.

Enumerator

UNCONFIGURED_POWER_DOWN	Power down mode.
POWER_ON	Power on mode.
RESET	Reset mode.
CONTINUOUS_HIGH_RES_MODE	Continuous high resolution mode.
CONTINUOUS_HIGH_RES_MODE_2	Continuous high resolution mode 2.
CONTINUOUS_LOW_RES_MODE	Continuous low resolution mode.
ONE_TIME_HIGH_RES_MODE	One-time high resolution mode.
ONE_TIME_HIGH_RES_MODE_2	One-time high resolution mode 2.
ONE_TIME_LOW_RES_MODE	One-time low resolution mode.

Definition at line 66 of file BH1750.h.

6.20 Sensors

Classes for handling sensor-related functions.

Classes

- class ISensor

  Abstract base class for sensors.
- class SensorWrapper

  Manages a collection of sensors.

Enumerations

- enum class SensorType : uint8_t { SensorType::NONE = 0x00 , SensorType::LIGHT = 0x01 , SensorType::ENVIRONMENT = 0x02 }

  Enumeration of sensor types.
- enum class SensorDataTypeIdentifier : uint8_t { SensorDataTypeIdentifier::NONE = 0x00 , SensorDataTypeIdentifier::LIGHT_LEVEL = 0x01 , SensorDataTypeIdentifier::TEMPE = 0x02 , SensorDataTypeIdentifier::HUMIDITY = 0x03 , SensorDataTypeIdentifier::PRESSURE = 0x04 }

  Enumeration of sensor data type identifiers.

Functions

- bool SensorWrapper::sensor_init (SensorType type, i2c_inst_t ∗i2c=nullptr)

  Initializes a sensor.
- bool SensorWrapper::sensor_configure (SensorType type, const std::map< std::string, std::string > &config)

  Configures a sensor.
- float SensorWrapper::sensor_read_data (SensorType sensorType, SensorDataTypeIdentifier dataType)

  Reads data from a sensor.
- ISensor ∗ SensorWrapper::get_sensor (SensorType type)

  Gets a sensor.
- std::vector< std::pair< SensorType, uint8_t > > SensorWrapper::scan_connected_sensors (i2c_inst_t ∗i2c)

  Scans for connected sensors.
- std::vector< std::pair< SensorType, uint8_t > > SensorWrapper::get_available_sensors ()

  Gets a list of available sensors.

6.20.1 Detailed Description

Classes for handling sensor-related functions.

6.20.2 Enumeration Type Documentation

6.20.2.1 SensorType

```
enum class SensorType : uint8_t [strong]
```

Enumeration of sensor types.

Defines the different types of sensors that can be managed.

Enumerator

NONE	No sensor.
LIGHT	Light sensor.
ENVIRONMENT	Environment sensor.

Definition at line 31 of file ISensor.h.

6.20.2.2 SensorDataTypeIdentifier

```
enum class SensorDataTypeIdentifier : uint8_t [strong]
```

Enumeration of sensor data type identifiers.

Defines the different types of data that can be read from a sensor.

Enumerator

NONE	No data.
LIGHT_LEVEL	Light level.
TEMPERATURE	Temperature.
HUMIDITY	Humidity.
PRESSURE	Pressure.

Definition at line 45 of file ISensor.h.

6.20.3 Function Documentation

6.20.3.1 sensor_init()

```
bool SensorWrapper::sensor_init (
            SensorType type,
            i2c_inst_t * i2c = nullptr)
```

Initializes a sensor.

Parameters

in	*type*	Sensor type to initialize.
in	*i2c*	I2C instance to use for communication.

Returns

True if initialization was successful, false otherwise.

Definition at line 27 of file ISensor.cpp.

6.20.3.2 sensor_configure()

```
bool SensorWrapper::sensor_configure (
            SensorType type,
            const std::map< std::string, std::string > & config)
```

Configures a sensor.

Parameters

in	*type*	Sensor type to configure.
in	*config*	A map of configuration parameters.

Returns

True if configuration was successful, false otherwise.

Definition at line 48 of file ISensor.cpp.

6.20.3.3 sensor_read_data()

```
float SensorWrapper::sensor_read_data (
            SensorType sensorType,
            SensorDataTypeIdentifier dataType)
```

Reads data from a sensor.

Parameters

in	*sensorType*	Sensor type to read from.
in	*dataType*	Data type to read.

Returns

The sensor data.

Definition at line 62 of file ISensor.cpp.

6.20.3.4 get_sensor()

```
ISensor * SensorWrapper::get_sensor (
            SensorType type)
```

Gets a sensor.

Parameters

in	*type*	Sensor type to get.

Returns

A pointer to the sensor.

Definition at line 75 of file ISensor.cpp.

6.20.3.5 scan_connected_sensors()

```
std::vector< std::pair< SensorType, uint8_t > > SensorWrapper::scan_connected_sensors (
            i2c_inst_t * i2c)
```

Scans for connected sensors.

Parameters

in	*i2c*	I2C instance to use for scanning.

Returns

A vector of pairs, where each pair contains a sensor type and its address.

Definition at line 85 of file ISensor.cpp.

6.20.3.6 get_available_sensors()

```
std::vector< std::pair< SensorType, uint8_t > > SensorWrapper::get_available_sensors ()
```

Gets a list of available sensors.

Returns

A vector of pairs, where each pair contains a sensor type and its address.

Definition at line 108 of file ISensor.cpp.

6.21 Storage

Classes and functions for managing file system operations.

Functions

- bool fs_init (void)
  Initializes the file system on the SD card.
- bool fs_stop (void)
  Unmounts the file system from the SD card.

6.21.1 Detailed Description

Classes and functions for managing file system operations.

6.21.2 Function Documentation

6.21.2.1 fs_init()

```
bool fs_init (
            void )
```

Initializes the file system on the SD card.

Returns

True if initialization was successful, false otherwise.

Mounts the littlefs file system on the SD card. If mounting fails, it formats the SD card with littlefs and then attempts to mount again.

Returns

True if initialization was successful, false otherwise.

Mounts the FAT file system on the SD card. If mounting fails, it formats the SD card with FAT and then attempts to mount again.

Definition at line 25 of file storage.cpp.

6.21.2.2 fs_stop()

```
bool fs_stop (
            void )
```

Unmounts the file system from the SD card.

Returns

True if unmounting was successful, false otherwise.

Definition at line 65 of file storage.cpp.

6.22 System State Manager

Classes for handling system state management.

Classes

- class SystemStateManager

  Manages the system state of the Kubisat firmware.

6.22.1 Detailed Description

Classes for handling system state management.

6.23 Telemetry Manager

Classes

- struct TelemetryRecord

  Structure representing a single telemetry data point.
- struct SensorDataRecord

  Structure representing a single sensor data point.
- class TelemetryManager

  Manages the collection, storage, and retrieval of telemetry data.

Macros

- #define TELEMETRY_CSV_PATH "/telemetry.csv"

  Path to the telemetry CSV file on storage media.
- #define SENSOR_DATA_CSV_PATH "/sensors.csv"

  Path to the sensor data CSV file on storage media.
- #define DEFAULT_SAMPLE_INTERVAL_MS 1000

  Default interval between telemetry samples in milliseconds (2 seconds)
- #define DEFAULT_FLUSH_THRESHOLD 10

  Default number of records to collect before flushing to storage.

Functions

- std::string TelemetryRecord::to_csv () const
  Converts the telemetry record to a CSV string.
- std::string SensorDataRecord::to_csv () const
  Converts the sensor data record to a CSV string.
- void TelemetryManager::collect_power_telemetry (TelemetryRecord &record)
  Collects power subsystem telemetry data.
- void TelemetryManager::emit_power_events (float battery_voltage, float charge_current_usb, float charge↩_current_solar)
  Emits power-related events based on current and voltage levels.
- void TelemetryManager::collect_gps_telemetry (TelemetryRecord &record)
  Collects GPS telemetry data.
- void TelemetryManager::collect_sensor_telemetry (SensorDataRecord &sensor_record)
  Collects sensor telemetry data.
- TelemetryManager::TelemetryManager ()
- bool TelemetryManager::init ()
  Initialize the telemetry system.
- bool TelemetryManager::collect_telemetry ()
  Collect telemetry data from sensors and power subsystems.
- bool TelemetryManager::flush_telemetry ()
  Save buffered telemetry data to storage.
- bool TelemetryManager::is_telemetry_collection_time (uint32_t current_time, uint32_t &last_collection_time)
  Check if it's time to collect telemetry based on interval.
- bool TelemetryManager::is_telemetry_flush_time (uint32_t &collection_counter)
  Check if it's time to flush telemetry buffer based on count.
- std::string TelemetryManager::get_last_telemetry_record_csv ()
  Gets the last telemetry record as a CSV string.
- std::string TelemetryManager::get_last_sensor_record_csv ()
  Gets the last sensor data record as a CSV string.

6.23.1 Detailed Description

6.23.2 Macro Definition Documentation

6.23.2.1 TELEMETRY_CSV_PATH

```
#define TELEMETRY_CSV_PATH "/telemetry.csv"
```

Path to the telemetry CSV file on storage media.

Definition at line 27 of file telemetry_manager.cpp.

6.23.2.2 SENSOR_DATA_CSV_PATH

```
#define SENSOR_DATA_CSV_PATH "/sensors.csv"
```

Path to the sensor data CSV file on storage media.

Definition at line 32 of file telemetry_manager.cpp.

6.23.2.3 DEFAULT_SAMPLE_INTERVAL_MS

```
#define DEFAULT_SAMPLE_INTERVAL_MS 1000
```

Default interval between telemetry samples in milliseconds (2 seconds)

Definition at line 37 of file telemetry_manager.cpp.

6.23.2.4 DEFAULT_FLUSH_THRESHOLD

```
#define DEFAULT_FLUSH_THRESHOLD 10
```

Default number of records to collect before flushing to storage.

Definition at line 42 of file telemetry_manager.cpp.

6.23.3 Function Documentation

6.23.3.1 to_csv() [1/2]

```
std::string TelemetryRecord::to_csv () const  [inline]
```

Converts the telemetry record to a CSV string.

Returns

A CSV string representing the telemetry record.

Definition at line 76 of file telemetry_manager.h.

6.23.3.2 to_csv() [2/2]

```
std::string SensorDataRecord::to_csv () const  [inline]
```

Converts the sensor data record to a CSV string.

Returns

A CSV string representing the sensor data record.

Definition at line 122 of file telemetry_manager.h.

6.23.3.3 collect_power_telemetry()

```
void TelemetryManager::collect_power_telemetry (
            TelemetryRecord & record)
```

Collects power subsystem telemetry data.

Parameters

out	*record*	The telemetry record to update with power data.

Definition at line 108 of file telemetry_manager.cpp.

6.23.3.4 emit_power_events()

```
void TelemetryManager::emit_power_events (
            float battery_voltage,
            float charge_current_usb,
            float charge_current_solar)
```

Emits power-related events based on current and voltage levels.

Parameters

in	*battery_voltage*	The current battery voltage.
in	*charge_current_usb*	The current USB charging current.
in	*charge_current_solar*	The current solar charging current.

Definition at line 123 of file telemetry_manager.cpp.

6.23.3.5 collect_gps_telemetry()

```
void TelemetryManager::collect_gps_telemetry (
            TelemetryRecord & record)
```

Collects GPS telemetry data.

Parameters

out	*record*	The telemetry record to update with GPS data.

Definition at line 172 of file telemetry_manager.cpp.

6.23.3.6 collect_sensor_telemetry()

```
void TelemetryManager::collect_sensor_telemetry (
            SensorDataRecord & sensor_record)
```

Collects sensor telemetry data.

Parameters

out	*sensor_record*	The sensor data record to update with sensor data.

Definition at line 218 of file telemetry_manager.cpp.

6.23.3.7 TelemetryManager()

```
TelemetryManager::TelemetryManager ()  [private]
```

Definition at line 44 of file telemetry_manager.cpp.

6.23.3.8 init()

```
bool TelemetryManager::init ()
```

Initialize the telemetry system.

Initializes the telemetry manager.

Returns

True if initialization was successful

Sets up the mutex for thread-safe buffer access and creates a telemetry CSV file with appropriate headers if it doesn't already exist

Returns

True if initialization was successful, false otherwise.

Initializes the telemetry mutex, checks if the SD card is mounted, and creates the telemetry and sensor data CSV files if they don't exist. Also writes the CSV headers to the files.

Definition at line 54 of file telemetry_manager.cpp.

6.23.3.9 collect_telemetry()

```
bool TelemetryManager::collect_telemetry ()
```

Collect telemetry data from sensors and power subsystems.

Returns

True if data was successfully collected

Reads data from power manager, sensors, and GPS and stores it in the telemetry buffer with proper mutex protection

Definition at line 233 of file telemetry_manager.cpp.

6.23.3.10 flush_telemetry()

```
bool TelemetryManager::flush_telemetry ()
```

Save buffered telemetry data to storage.

Save buffered telemetry and sensor data to storage.

Returns

True if data was successfully saved

Writes all records from the telemetry buffer to the CSV file and clears the buffer after successful writing

Returns

True if data was successfully saved

Writes all records from the telemetry and sensor data buffers to their respective CSV files and clears the buffers after successful writing

Definition at line 275 of file telemetry_manager.cpp.

6.23.3.11 is_telemetry_collection_time()

```
bool TelemetryManager::is_telemetry_collection_time (
            uint32_t current_time,
            uint32_t & last_collection_time)
```

Check if it's time to collect telemetry based on interval.

Parameters

current_time	Current system time in milliseconds
last_collection_time	Previous collection time in milliseconds

Returns

True if collection interval has passed

Updates last_collection_time if the interval has passed

Definition at line 331 of file telemetry_manager.cpp.

6.23.3.12 is_telemetry_flush_time()

```
bool TelemetryManager::is_telemetry_flush_time (
            uint32_t & collection_counter)
```

Check if it's time to flush telemetry buffer based on count.

Parameters

collection_counter	Current collection counter

Returns

True if flush threshold has been reached

Resets collection_counter to zero if the threshold has been reached

Definition at line 347 of file telemetry_manager.cpp.

6.23.3.13 get_last_telemetry_record_csv()

```
std::string TelemetryManager::get_last_telemetry_record_csv ()
```

Gets the last telemetry record as a CSV string.

Returns

A CSV string representing the last telemetry record, or an empty string if no data is available.

Definition at line 360 of file telemetry_manager.cpp.

6.23.3.14 get_last_sensor_record_csv()

```
std::string TelemetryManager::get_last_sensor_record_csv ()
```

Gets the last sensor data record as a CSV string.

Returns

A CSV string representing the last sensor data record, or an empty string if no data is available.

Definition at line 380 of file telemetry_manager.cpp.

Chapter 7

Class Documentation

7.1 BH1750 Class Reference

Class to interface with the BH1750 light sensor.

`#include <BH1750.h>`

Public Types

- enum class Mode : uint8_t {
  Mode::UNCONFIGURED_POWER_DOWN = 0x00 , Mode::POWER_ON = 0x01 , Mode::RESET = 0x07 ,
  Mode::CONTINUOUS_HIGH_RES_MODE = 0x10 ,
  Mode::CONTINUOUS_HIGH_RES_MODE_2 = 0x11 , Mode::CONTINUOUS_LOW_RES_MODE = 0x13 ,
  Mode::ONE_TIME_HIGH_RES_MODE = 0x20 , Mode::ONE_TIME_HIGH_RES_MODE_2 = 0x21 ,
  Mode::ONE_TIME_LOW_RES_MODE = 0x23 }

  Enumeration of measurement modes for the BH1750 sensor.

Public Member Functions

- BH1750 (i2c_inst_t ∗i2c, uint8_t addr=0x23)

  Constructor for the BH1750 class.
- bool begin (Mode mode=Mode::CONTINUOUS_HIGH_RES_MODE)

  Initializes the BH1750 sensor.
- bool configure (Mode mode)

  Configures the BH1750 sensor with the specified mode.
- float get_light_level ()

  Reads the light level from the BH1750 sensor.

Private Member Functions

- void write8 (uint8_t data)

  Writes a single byte of data to the BH1750 sensor.

Private Attributes

- uint8_t _i2c_addr

  I2C address of the BH1750 sensor.
- i2c_inst_t * i2c_port_

  Pointer to the I2C interface.

7.1.1 Detailed Description

Class to interface with the BH1750 light sensor.

Definition at line 60 of file BH1750.h.

7.1.2 Member Data Documentation

7.1.2.1 _i2c_addr

```
uint8_t BH1750::_i2c_addr  [private]
```

I2C address of the BH1750 sensor.

Definition at line 122 of file BH1750.h.

7.1.2.2 i2c_port_

```
i2c_inst_t* BH1750::i2c_port_  [private]
```

Pointer to the I2C interface.

Definition at line 124 of file BH1750.h.

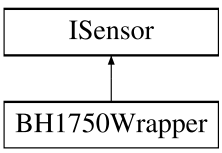
The documentation for this class was generated from the following files:

- lib/sensors/BH1750/BH1750.h
- lib/sensors/BH1750/BH1750.cpp

7.2 BH1750Wrapper Class Reference

```
#include <BH1750_WRAPPER.h>
```

Inheritance diagram for BH1750Wrapper:



Public Member Functions

- BH1750Wrapper (i2c_inst_t ∗i2c)
- BH1750Wrapper ()
- int get_i2c_addr ()
- bool init () override

  Initializes the sensor.
- float read_data (SensorDataTypeIdentifier type) override

  Reads data from the sensor.
- bool is_initialized () const override

  Checks if the sensor is initialized.
- SensorType get_type () const override

  Gets the sensor type.
- bool configure (const std::map< std::string, std::string > &config)

  Configures the sensor.
- uint8_t get_address () const override

  Gets the I2C address of the sensor.

Public Member Functions inherited from ISensor

- virtual ∼ISensor ()=default

  Virtual destructor.

Private Attributes

- BH1750 sensor_
- bool initialized_ = false

7.2.1 Detailed Description

Definition at line 9 of file BH1750_WRAPPER.h.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 BH1750Wrapper() [1/2]

```
BH1750Wrapper::BH1750Wrapper (
            i2c_inst_t * i2c)
```

Definition at line 5 of file BH1750_WRAPPER.cpp.

7.2.2.2 BH1750Wrapper() [2/2]

```
BH1750Wrapper::BH1750Wrapper ()
```

7.2.3 Member Function Documentation

7.2.3.1 get_i2c_addr()

```
int BH1750Wrapper::get_i2c_addr ()
```

7.2.3.2 init()

```
bool BH1750Wrapper::init () [override], [virtual]
```

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implements ISensor.

Definition at line 9 of file BH1750_WRAPPER.cpp.

7.2.3.3 read_data()

```
float BH1750Wrapper::read_data (
            SensorDataTypeIdentifier type) [override], [virtual]
```

Reads data from the sensor.

Parameters

in	*type*	Data type to read.

Returns

The sensor data.

Implements ISensor.

Definition at line 14 of file BH1750_WRAPPER.cpp.

7.2.3.4 is_initialized()

```
bool BH1750Wrapper::is_initialized () const [override], [virtual]
```

Checks if the sensor is initialized.

Returns

True if the sensor is initialized, false otherwise.

Implements ISensor.

Definition at line 21 of file BH1750_WRAPPER.cpp.

7.2.3.5 get_type()

```
SensorType BH1750Wrapper::get_type () const  [override], [virtual]
```

Gets the sensor type.

Returns

The sensor type.

Implements ISensor.

Definition at line 25 of file BH1750_WRAPPER.cpp.

7.2.3.6 configure()

```
bool BH1750Wrapper::configure (
            const std::map< std::string, std::string > & config)  [virtual]
```

Configures the sensor.

Parameters

in	*config*	A map of configuration parameters.

Returns

True if configuration was successful, false otherwise.

Implements ISensor.

Definition at line 29 of file BH1750_WRAPPER.cpp.

7.2.3.7 get_address()

```
uint8_t BH1750Wrapper::get_address () const  [inline], [override], [virtual]
```

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

Implements ISensor.

Definition at line 25 of file BH1750_WRAPPER.h.

7.2.4 Member Data Documentation

7.2.4.1 sensor_

`BH1750 BH1750Wrapper::sensor_ [private]`

Definition at line 11 of file BH1750_WRAPPER.h.

7.2.4.2 initialized_

`bool BH1750Wrapper::initialized_ = false [private]`

Definition at line 12 of file BH1750_WRAPPER.h.

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

- lib/sensors/BH1750/BH1750_WRAPPER.h
- lib/sensors/BH1750/BH1750_WRAPPER.cpp

7.3 BME280 Class Reference

Class to interface with the BME280 environmental sensor.

`#include <BME280.h>`

Public Types

- enum { ADDR_SDO_LOW = 0x76 , ADDR_SDO_HIGH = 0x77 }

  I2C Address Options for the BME280 sensor.
- enum class Oversampling : uint8_t {
  OSR_X0 = 0x00 , OSR_X1 = 0x01 , OSR_X2 = 0x02 , OSR_X4 = 0x03 ,
  OSR_X8 = 0x04 , OSR_X16 = 0x05 }

  Enum class for oversampling settings.

Public Member Functions

- BME280 (i2c_inst_t ∗i2cPort, uint8_t address=ADDR_SDO_LOW)

  Constructor for the BME280 class.
- bool init ()

  Initializes the BME280 sensor.
- void reset ()

  Resets the BME280 sensor.
- bool read_raw_all (int32_t ∗temperature, int32_t ∗pressure, int32_t ∗humidity)

  Reads all raw data from the sensor.
- float convert_temperature (int32_t temp_raw) const

  Converts raw temperature data to degrees Celsius.
- float convert_pressure (int32_t pressure_raw) const

  Converts raw pressure data to hectopascals (hPa).
- float convert_humidity (int32_t humidity_raw) const

  Converts raw humidity data to relative humidity (%).

Private Types

- enum {
REG_CONFIG = 0xF5 , REG_CTRL_MEAS = 0xF4 , REG_CTRL_HUM = 0xF2 , REG_RESET = 0xE0 ,
REG_PRESSURE_MSB = 0xF7 , REG_TEMPERATURE_MSB = 0xFA , REG_HUMIDITY_MSB = 0xFD ,
REG_DIG_T1_LSB = 0x88 ,
REG_DIG_T1_MSB = 0x89 , REG_DIG_T2_LSB = 0x8A , REG_DIG_T2_MSB = 0x8B , REG_DIG_T3_LSB = 0x8C ,
REG_DIG_T3_MSB = 0x8D , REG_DIG_P1_LSB = 0x8E , REG_DIG_P1_MSB = 0x8F , REG_DIG_P2_LSB = 0x90 ,
REG_DIG_P2_MSB = 0x91 , REG_DIG_P3_LSB = 0x92 , REG_DIG_P3_MSB = 0x93 , REG_DIG_P4_LSB = 0x94 ,
REG_DIG_P4_MSB = 0x95 , REG_DIG_P5_LSB = 0x96 , REG_DIG_P5_MSB = 0x97 , REG_DIG_P6_LSB = 0x98 ,
REG_DIG_P6_MSB = 0x99 , REG_DIG_P7_LSB = 0x9A , REG_DIG_P7_MSB = 0x9B , REG_DIG_P8_LSB = 0x9C ,
REG_DIG_P8_MSB = 0x9D , REG_DIG_P9_LSB = 0x9E , REG_DIG_P9_MSB = 0x9F , REG_DIG_H1 = 0xA1 ,
REG_DIG_H2 = 0xE1 , REG_DIG_H3 = 0xE3 , REG_DIG_H4 = 0xE4 , REG_DIG_H5 = 0xE5 ,
REG_DIG_H6 = 0xE7 }

    Register Definitions for the BME280 sensor.
- enum { HUMIDITY_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16) , TEMPERATURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16) , PRESSURE_OVERSAMPLING = static_cast<uint8_↩
t>(Oversampling::OSR_X16) , NORMAL_MODE = 0xB7 }

    Sensor settings.
- enum { NUM_CALIB_PARAMS = 26 , NUM_HUM_CALIB_PARAMS = 7 }

    Calibration data length.

Private Member Functions

- bool write_register (uint8_t reg, uint8_t value)

    Helper function for I2C writes.
- bool read_register (uint8_t reg, uint8_t *data)

    Helper function for I2C reads.
- bool read_register (uint8_t reg, uint8_t *data, size_t len)

    Helper function for I2C reads with a specified length.
- bool configure_sensor ()

    Configures the sensor with default settings.
- bool get_calibration_parameters ()

    Retrieves the calibration parameters from the sensor.

Private Attributes

- i2c_inst_t * i2c_port

    Pointer to the I2C interface.
- uint8_t device_addr

    I2C device address.
- BME280CalibParam calib_params

    Calibration parameters for the sensor.
- bool initialized_

    Initialization status of the sensor.
- int32_t t_fine

    Fine temperature parameter needed for compensation.

7.3.1 Detailed Description

Class to interface with the BME280 environmental sensor.

This class provides methods to initialize the sensor, read raw data, convert raw data to physical units (temperature, pressure, humidity), and configure the sensor's operating mode.

Definition at line 71 of file BME280.h.

7.3.2 Member Enumeration Documentation

7.3.2.1 anonymous enum

```
anonymous enum
```

I2C Address Options for the BME280 sensor.

Enumerator

ADDR_SDO_LOW	I2C address when SDO pin is low.
ADDR_SDO_HIGH	I2C address when SDO pin is high.

Definition at line 76 of file BME280.h.

7.3.2.2 Oversampling

```
enum class BME280::Oversampling : uint8_t [strong]
```

Enum class for oversampling settings.

These settings determine the number of measurements that are averaged to reduce noise and improve the accuracy of the sensor readings.

Enumerator

OSR_X0	No oversampling.
OSR_X1	1x oversampling
OSR_X2	2x oversampling
OSR_X4	4x oversampling
OSR_X8	8x oversampling
OSR_X16	16x oversampling

Definition at line 89 of file BME280.h.

7.3.2.3 anonymous enum

```
anonymous enum [private]
```

Register Definitions for the BME280 sensor.

Enumerator

REG_CONFIG	Configuration register.
REG_CTRL_MEAS	Control measurement register.
REG_CTRL_HUM	Control humidity register.
REG_RESET	Reset register.
REG_PRESSURE_MSB	Pressure data MSB.
REG_TEMPERATURE_MSB	Temperature data MSB.
REG_HUMIDITY_MSB	Humidity data MSB.
REG_DIG_T1_LSB	Calibration data LSB.
REG_DIG_T1_MSB	Calibration data MSB.
REG_DIG_T2_LSB	Calibration data LSB.
REG_DIG_T2_MSB	Calibration data MSB.
REG_DIG_T3_LSB	Calibration data LSB.
REG_DIG_T3_MSB	Calibration data MSB.
REG_DIG_P1_LSB	Calibration data LSB.
REG_DIG_P1_MSB	Calibration data MSB.
REG_DIG_P2_LSB	Calibration data LSB.
REG_DIG_P2_MSB	Calibration data MSB.
REG_DIG_P3_LSB	Calibration data LSB.
REG_DIG_P3_MSB	Calibration data MSB.
REG_DIG_P4_LSB	Calibration data LSB.
REG_DIG_P4_MSB	Calibration data MSB.
REG_DIG_P5_LSB	Calibration data LSB.
REG_DIG_P5_MSB	Calibration data MSB.
REG_DIG_P6_LSB	Calibration data LSB.
REG_DIG_P6_MSB	Calibration data MSB.
REG_DIG_P7_LSB	Calibration data LSB.
REG_DIG_P7_MSB	Calibration data MSB.
REG_DIG_P8_LSB	Calibration data LSB.
REG_DIG_P8_MSB	Calibration data MSB.
REG_DIG_P9_LSB	Calibration data LSB.
REG_DIG_P9_MSB	Calibration data MSB.
REG_DIG_H1	Humidity calibration data.
REG_DIG_H2	Humidity calibration data.
REG_DIG_H3	Humidity calibration data.
REG_DIG_H4	Humidity calibration data.
REG_DIG_H5	Humidity calibration data.
REG_DIG_H6	Humidity calibration data.

Definition at line 207 of file BME280.h.

7.3.2.4 anonymous enum

```
anonymous enum  [private]
```

Sensor settings.

Enumerator

HUMIDITY_OVERSAMPLING	Humidity oversampling setting.
TEMPERATURE_OVERSAMPLING	Temperature oversampling setting.
PRESSURE_OVERSAMPLING	Pressure oversampling setting.
NORMAL_MODE	Normal mode setting.

Definition at line 256 of file BME280.h.

7.3.2.5 anonymous enum

```
anonymous enum  [private]
```

Calibration data length.

Enumerator

NUM_CALIB_PARAMS	Number of calibration parameters.
NUM_HUM_CALIB_PARAMS	Number of humidity calibration parameters.

Definition at line 266 of file BME280.h.

7.3.3 Constructor & Destructor Documentation

7.3.3.1 BME280()

```
BME280::BME280 (
            i2c_inst_t * i2cPort,
            uint8_t address = ADDR_SDO_LOW)
```

Constructor for the BME280 class.

Parameters

i2cPort	Pointer to the I2C interface.
address	I2C address of the BME280 sensor (default: ADDR_SDO_LOW).

Definition at line 24 of file BME280.cpp.

7.3.4 Member Function Documentation

7.3.4.1 init()

```
bool BME280::init ()
```

Initializes the BME280 sensor.

Returns

True if initialization was successful, false otherwise.

Definition at line 32 of file BME280.cpp.

7.3.4.2 reset()

```
void BME280::reset ()
```

Resets the BME280 sensor.

Definition at line 70 of file BME280.cpp.

7.3.4.3 read_raw_all()

```
bool BME280::read_raw_all (
            int32_t * temperature,
            int32_t * pressure,
            int32_t * humidity)
```

Reads all raw data from the sensor.

Parameters

temperature	Pointer to store the raw temperature value.
pressure	Pointer to store the raw pressure value.
humidity	Pointer to store the raw humidity value.

Returns

True if the data was read successfully, false otherwise.

Definition at line 82 of file BME280.cpp.

7.3.4.4 convert_temperature()

```
float BME280::convert_temperature (
            int32_t temp_raw) const
```

Converts raw temperature data to degrees Celsius.

Parameters

temp_raw	Raw temperature value.

Returns

Temperature in degrees Celsius.

Definition at line 119 of file BME280.cpp.

7.3.4.5 convert_pressure()

```
float BME280::convert_pressure (
            int32_t pressure_raw) const
```

Converts raw pressure data to hectopascals (hPa).

Parameters

pressure_raw	Raw pressure value.

Returns

Pressure in hPa.

Definition at line 133 of file BME280.cpp.

7.3.4.6 convert_humidity()

```
float BME280::convert_humidity (
            int32_t humidity_raw) const
```

Converts raw humidity data to relative humidity (%).

Parameters

humidity_raw	Raw humidity value.

Returns

Relative humidity in %.

Definition at line 159 of file BME280.cpp.

7.3.4.7 write_register()

```
bool BME280::write_register (
            uint8_t reg,
            uint8_t value) [private]
```

Helper function for I2C writes.

Parameters

reg	Register address to write to.
value	Value to write to the register.

Returns

True if the write was successful, false otherwise.

Definition at line 250 of file BME280.cpp.

7.3.4.8 read_register() [1/2]

```
bool BME280::read_register (
            uint8_t reg,
            uint8_t * data) [private]
```

Helper function for I2C reads.

Parameters

reg	Register address to read from.
data	Pointer to store the read data.

Returns

True if the read was successful, false otherwise.

Definition at line 278 of file BME280.cpp.

7.3.4.9 read_register() [2/2]

```
bool BME280::read_register (
            uint8_t reg,
            uint8_t * data,
            size_t len)  [private]
```

Helper function for I2C reads with a specified length.

Parameters

reg	Register address to read from.
data	Pointer to store the read data.
len	Number of bytes to read.

Returns

True if the read was successful, false otherwise.

Definition at line 263 of file BME280.cpp.

7.3.4.10 configure_sensor()

```
bool BME280::configure_sensor ()  [private]
```

Configures the sensor with default settings.

Returns

True if the configuration was successful, false otherwise.

Definition at line 222 of file BME280.cpp.

7.3.4.11 get_calibration_parameters()

`bool BME280::get_calibration_parameters () [private]`

Retrieves the calibration parameters from the sensor.

Returns

True if the parameters were read successfully, false otherwise.

Definition at line 175 of file BME280.cpp.

7.3.5 Member Data Documentation

7.3.5.1 i2c_port

`i2c_inst_t* BME280::i2c_port [private]`

Pointer to the I2C interface.

Definition at line 191 of file BME280.h.

7.3.5.2 device_addr

`uint8_t BME280::device_addr [private]`

I2C device address.

Definition at line 193 of file BME280.h.

7.3.5.3 calib_params

`BME280CalibParam BME280::calib_params [private]`

Calibration parameters for the sensor.

Definition at line 196 of file BME280.h.

7.3.5.4 initialized_

`bool BME280::initialized_ [private]`

Initialization status of the sensor.

Definition at line 199 of file BME280.h.

7.3.5.5 t_fine

`int32_t BME280::t_fine [mutable], [private]`

Fine temperature parameter needed for compensation.

Definition at line 202 of file BME280.h.

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

- lib/sensors/BME280/BME280.h
- lib/sensors/BME280/BME280.cpp

7.4 BME280CalibParam Struct Reference

Structure to hold the BME280 calibration parameters.

`#include <BME280.h>`

Public Attributes

- uint16_t dig_t1
  Temperature calibration parameter 1.
- int16_t dig_t2
  Temperature calibration parameter 2.
- int16_t dig_t3
  Temperature calibration parameter 3.
- uint16_t dig_p1
  Pressure calibration parameter 1.
- int16_t dig_p2
  Pressure calibration parameter 2.
- int16_t dig_p3
  Pressure calibration parameter 3.
- int16_t dig_p4
  Pressure calibration parameter 4.
- int16_t dig_p5
  Pressure calibration parameter 5.
- int16_t dig_p6
  Pressure calibration parameter 6.
- int16_t dig_p7
  Pressure calibration parameter 7.
- int16_t dig_p8
  Pressure calibration parameter 8.
- int16_t dig_p9
  Pressure calibration parameter 9.
- uint8_t dig_h1
  Humidity calibration parameter 1.
- int16_t dig_h2
  Humidity calibration parameter 2.
- uint8_t dig_h3
  Humidity calibration parameter 3.
- int16_t dig_h4
  Humidity calibration parameter 4.
- int16_t dig_h5
  Humidity calibration parameter 5.
- int8_t dig_h6
  Humidity calibration parameter 6.

7.4.1 Detailed Description

Structure to hold the BME280 calibration parameters.

These parameters are unique to each sensor and are used to compensate for manufacturing variations and improve the accuracy of the sensor readings.

Definition at line 23 of file BME280.h.

7.4.2 Member Data Documentation

7.4.2.1 dig_t1

`uint16_t BME280CalibParam::dig_t1`

Temperature calibration parameter 1.

Definition at line 25 of file BME280.h.

7.4.2.2 dig_t2

`int16_t BME280CalibParam::dig_t2`

Temperature calibration parameter 2.

Definition at line 27 of file BME280.h.

7.4.2.3 dig_t3

`int16_t BME280CalibParam::dig_t3`

Temperature calibration parameter 3.

Definition at line 29 of file BME280.h.

7.4.2.4 dig_p1

`uint16_t BME280CalibParam::dig_p1`

Pressure calibration parameter 1.

Definition at line 32 of file BME280.h.

7.4.2.5 dig_p2

`int16_t BME280CalibParam::dig_p2`

Pressure calibration parameter 2.

Definition at line 34 of file BME280.h.

7.4.2.6 dig_p3

```
int16_t BME280CalibParam::dig_p3
```

Pressure calibration parameter 3.

Definition at line 36 of file BME280.h.

7.4.2.7 dig_p4

```
int16_t BME280CalibParam::dig_p4
```

Pressure calibration parameter 4.

Definition at line 38 of file BME280.h.

7.4.2.8 dig_p5

```
int16_t BME280CalibParam::dig_p5
```

Pressure calibration parameter 5.

Definition at line 40 of file BME280.h.

7.4.2.9 dig_p6

```
int16_t BME280CalibParam::dig_p6
```

Pressure calibration parameter 6.

Definition at line 42 of file BME280.h.

7.4.2.10 dig_p7

```
int16_t BME280CalibParam::dig_p7
```

Pressure calibration parameter 7.

Definition at line 44 of file BME280.h.

7.4.2.11 dig_p8

```
int16_t BME280CalibParam::dig_p8
```

Pressure calibration parameter 8.

Definition at line 46 of file BME280.h.

7.4.2.12 dig_p9

`int16_t BME280CalibParam::dig_p9`

Pressure calibration parameter 9.

Definition at line 48 of file BME280.h.

7.4.2.13 dig_h1

`uint8_t BME280CalibParam::dig_h1`

Humidity calibration parameter 1.

Definition at line 51 of file BME280.h.

7.4.2.14 dig_h2

`int16_t BME280CalibParam::dig_h2`

Humidity calibration parameter 2.

Definition at line 53 of file BME280.h.

7.4.2.15 dig_h3

`uint8_t BME280CalibParam::dig_h3`

Humidity calibration parameter 3.

Definition at line 55 of file BME280.h.

7.4.2.16 dig_h4

`int16_t BME280CalibParam::dig_h4`

Humidity calibration parameter 4.

Definition at line 57 of file BME280.h.

7.4.2.17 dig_h5

`int16_t BME280CalibParam::dig_h5`

Humidity calibration parameter 5.

Definition at line 59 of file BME280.h.

7.4.2.18 dig_h6

```
int8_t BME280CalibParam::dig_h6
```

Humidity calibration parameter 6.

Definition at line 61 of file BME280.h.

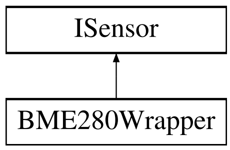
The documentation for this struct was generated from the following file:

- lib/sensors/BME280/BME280.h

7.5 BME280Wrapper Class Reference

```
#include <BME280_WRAPPER.h>
```

Inheritance diagram for BME280Wrapper:



Public Member Functions

- BME280Wrapper (i2c_inst_t ∗i2c)
- bool init () override
  Initializes the sensor.
- float read_data (SensorDataTypeIdentifier type) override
  Reads data from the sensor.
- bool is_initialized () const override
  Checks if the sensor is initialized.
- SensorType get_type () const override
  Gets the sensor type.
- bool configure (const std::map< std::string, std::string > &config) override
  Configures the sensor.
- uint8_t get_address () const override
  Gets the I2C address of the sensor.

Public Member Functions inherited from ISensor

- virtual ∼ISensor ()=default
  Virtual destructor.

Private Attributes

- BME280 sensor_
- bool initialized_ = false

7.5.1 Detailed Description

Definition at line 8 of file BME280_WRAPPER.h.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 BME280Wrapper()

```
BME280Wrapper::BME280Wrapper (
            i2c_inst_t * i2c)
```

Definition at line 3 of file BME280_WRAPPER.cpp.

7.5.3 Member Function Documentation

7.5.3.1 init()

```
bool BME280Wrapper::init ()  [override], [virtual]
```

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implements ISensor.

Definition at line 5 of file BME280_WRAPPER.cpp.

7.5.3.2 read_data()

```
float BME280Wrapper::read_data (
            SensorDataTypeIdentifier type)  [override], [virtual]
```

Reads data from the sensor.

Parameters

in	*type*	Data type to read.

Returns

The sensor data.

Implements ISensor.

Definition at line 10 of file BME280_WRAPPER.cpp.

7.5.3.3 is_initialized()

```
bool BME280Wrapper::is_initialized () const  [override], [virtual]
```

Checks if the sensor is initialized.

Returns

True if the sensor is initialized, false otherwise.

Implements ISensor.

Definition at line 26 of file BME280_WRAPPER.cpp.

7.5.3.4 get_type()

```
SensorType BME280Wrapper::get_type () const  [override], [virtual]
```

Gets the sensor type.

Returns

The sensor type.

Implements ISensor.

Definition at line 30 of file BME280_WRAPPER.cpp.

7.5.3.5 configure()

```
bool BME280Wrapper::configure (
            const std::map< std::string, std::string > & config)  [override], [virtual]
```

Configures the sensor.

Parameters

in	*config*	A map of configuration parameters.

Returns

True if configuration was successful, false otherwise.

Implements ISensor.

Definition at line 34 of file BME280_WRAPPER.cpp.

7.5.3.6 get_address()

```
uint8_t BME280Wrapper::get_address () const [inline], [override], [virtual]
```

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

Implements ISensor.

Definition at line 22 of file BME280_WRAPPER.h.

7.5.4 Member Data Documentation

7.5.4.1 sensor_

```
BME280 BME280Wrapper::sensor_ [private]
```

Definition at line 10 of file BME280_WRAPPER.h.

7.5.4.2 initialized_

```
bool BME280Wrapper::initialized_ = false [private]
```

Definition at line 11 of file BME280_WRAPPER.h.

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

- lib/sensors/BME280/BME280_WRAPPER.h
- lib/sensors/BME280/BME280_WRAPPER.cpp

7.6 INA3221::conf_reg_t Struct Reference

Configuration register bit fields.

Public Attributes

- uint16_t mode_shunt_en:1
- uint16_t mode_bus_en:1
- uint16_t mode_continious_en:1
- uint16_t shunt_conv_time:3
- uint16_t bus_conv_time:3
- uint16_t avg_mode:3
- uint16_t ch3_en:1
- uint16_t ch2_en:1
- uint16_t ch1_en:1
- uint16_t reset:1

7.6.1 Detailed Description

Configuration register bit fields.

Definition at line 101 of file INA3221.h.

7.6.2 Member Data Documentation

7.6.2.1 mode_shunt_en

`uint16_t INA3221::conf_reg_t::mode_shunt_en`

Definition at line 102 of file INA3221.h.

7.6.2.2 mode_bus_en

`uint16_t INA3221::conf_reg_t::mode_bus_en`

Definition at line 103 of file INA3221.h.

7.6.2.3 mode_continious_en

`uint16_t INA3221::conf_reg_t::mode_continious_en`

Definition at line 104 of file INA3221.h.

7.6.2.4 shunt_conv_time

`uint16_t INA3221::conf_reg_t::shunt_conv_time`

Definition at line 105 of file INA3221.h.

7.6.2.5 bus_conv_time

`uint16_t INA3221::conf_reg_t::bus_conv_time`

Definition at line 106 of file INA3221.h.

7.6.2.6 avg_mode

`uint16_t INA3221::conf_reg_t::avg_mode`

Definition at line 107 of file INA3221.h.

7.6.2.7 ch3_en

```
uint16_t INA3221::conf_reg_t::ch3_en
```

Definition at line 108 of file INA3221.h.

7.6.2.8 ch2_en

```
uint16_t INA3221::conf_reg_t::ch2_en
```

Definition at line 109 of file INA3221.h.

7.6.2.9 ch1_en

```
uint16_t INA3221::conf_reg_t::ch1_en
```

Definition at line 110 of file INA3221.h.

7.6.2.10 reset

```
uint16_t INA3221::conf_reg_t::reset
```

Definition at line 111 of file INA3221.h.

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

- lib/powerman/INA3221/INA3221.h

7.7 DS3231 Class Reference

Class for interfacing with the DS3231 real-time clock.

```
#include <DS3231.h>
```

Public Member Functions

- DS3231 (i2c_inst_t *i2c_instance)
  Constructor for the DS3231 class.
- int set_time (ds3231_data_t *data)
  Sets the time on the DS3231 clock.
- int get_time (ds3231_data_t *data)
  Gets the current time from the DS3231 clock.
- int read_temperature (float *resolution)
  Reads the current temperature from the DS3231.
- int set_unix_time (time_t unix_time)
  Sets the time using a Unix timestamp.
- time_t get_unix_time ()
  Gets the current time as a Unix timestamp.
- int clock_enable ()
  Enables the DS3231 clock oscillator.
- int16_t get_timezone_offset () const
  Gets the current timezone offset.
- void set_timezone_offset (int16_t offset_minutes)
  Sets the timezone offset.
- uint32_t get_clock_sync_interval () const
  Gets the clock synchronization interval.
- void set_clock_sync_interval (uint32_t interval_minutes)
  Sets the clock synchronization interval.
- time_t get_last_sync_time () const
  Gets the timestamp of the last clock synchronization.
- void update_last_sync_time ()
  Updates the last sync time to current time.
- time_t get_local_time ()
  Gets the current local time (including timezone offset)
- bool is_sync_needed ()
  Checks if clock synchronization is needed.
- bool sync_clock_with_gps ()
  Synchronizes clock with GPS data.

Static Public Member Functions

- static DS3231 & get_instance ()
  Gets the singleton instance of the DS3231 class.

Private Member Functions

- DS3231 ()
  Constructor for the DS3231 class.
- DS3231 (const DS3231 &)=delete
- DS3231 & operator= (const DS3231 &)=delete
- int i2c_read_reg (uint8_t reg_addr, size_t length, uint8_t *data)
  Reads data from a specific register on the DS3231.
- int i2c_write_reg (uint8_t reg_addr, size_t length, uint8_t *data)
  Writes data to a specific register on the DS3231.
- uint8_t bin_to_bcd (const uint8_t data)
  Converts binary value to BCD (Binary Coded Decimal)
- uint8_t bcd_to_bin (const uint8_t bcd)
  Converts BCD (Binary Coded Decimal) to binary value.

Private Attributes

- i2c_inst_t ∗ i2c
- uint8_t ds3231_addr
- recursive_mutex_t clock_mutex_
- int16_t timezone_offset_minutes_ = 60
- uint32_t sync_interval_minutes_ = 1440
- time_t last_sync_time_ = 0

7.7.1 Detailed Description

Class for interfacing with the DS3231 real-time clock.

This class provides methods to set and get time from a DS3231 RTC module, handle timezone offsets, perform clock synchronization, and more.

Definition at line 108 of file DS3231.h.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 DS3231() [1/2]

```
DS3231::DS3231 (
            i2c_inst_t * i2c_instance)
```

Constructor for the DS3231 class.

Parameters

in	*i2c_instance*	Pointer to the I2C instance to use

7.7.2.2 DS3231() [2/2]

```
DS3231::DS3231 (
            const DS3231 & )  [private], [delete]
```

7.7.3 Member Function Documentation

7.7.3.1 operator=()

```
DS3231 & DS3231::operator= (
            const DS3231 & )  [private], [delete]
```

7.7.4 Member Data Documentation

7.7.4.1 i2c

```
i2c_inst_t* DS3231::i2c  [private]
```

Definition at line 232 of file DS3231.h.

7.7.4.2 ds3231_addr

```
uint8_t DS3231::ds3231_addr  [private]
```

Definition at line 233 of file DS3231.h.

7.7.4.3 clock_mutex_

```
recursive_mutex_t DS3231::clock_mutex_  [private]
```

Definition at line 234 of file DS3231.h.

7.7.4.4 timezone_offset_minutes_

```
int16_t DS3231::timezone_offset_minutes_ = 60  [private]
```

Definition at line 235 of file DS3231.h.

7.7.4.5 sync_interval_minutes_

```
uint32_t DS3231::sync_interval_minutes_ = 1440  [private]
```

Definition at line 236 of file DS3231.h.

7.7.4.6 last_sync_time_

```
time_t DS3231::last_sync_time_ = 0  [private]
```

Definition at line 237 of file DS3231.h.

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

- lib/clock/DS3231.h
- lib/clock/DS3231.cpp

7.8 ds3231_data_t Struct Reference

Structure to hold time and date information from DS3231.

```
#include <DS3231.h>
```

Public Attributes

- uint8_t seconds

  Seconds (0-59)
- uint8_t minutes

  Minutes (0-59)
- uint8_t hours

  Hours (0-23)
- uint8_t day

  Day of the week (1-7)
- uint8_t date

  Date (1-31)
- uint8_t month

  Month (1-12)
- uint8_t year

  Year (0-99)
- bool century

  Century flag (0-1)

7.8.1 Detailed Description

Structure to hold time and date information from DS3231.

Definition at line 90 of file DS3231.h.

7.8.2 Member Data Documentation

7.8.2.1 seconds

```
uint8_t ds3231_data_t::seconds
```

Seconds (0-59)

Definition at line 91 of file DS3231.h.

7.8.2.2 minutes

```
uint8_t ds3231_data_t::minutes
```

Minutes (0-59)

Definition at line 92 of file DS3231.h.

7.8.2.3 hours

```
uint8_t ds3231_data_t::hours
```

Hours (0-23)

Definition at line 93 of file DS3231.h.

7.8.2.4 day

```
uint8_t ds3231_data_t::day
```

Day of the week (1-7)

Definition at line 94 of file DS3231.h.

7.8.2.5 date

```
uint8_t ds3231_data_t::date
```

Date (1-31)

Definition at line 95 of file DS3231.h.

7.8.2.6 month

```
uint8_t ds3231_data_t::month
```

Month (1-12)

Definition at line 96 of file DS3231.h.

7.8.2.7 year

```
uint8_t ds3231_data_t::year
```

Year (0-99)

Definition at line 97 of file DS3231.h.

7.8.2.8 century

```
bool ds3231_data_t::century
```

Century flag (0-1)

Definition at line 98 of file DS3231.h.

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

- lib/clock/DS3231.h

7.9 EventEmitter Class Reference

Provides a simple interface for emitting events.

```
#include <event_manager.h>
```

Static Public Member Functions

- template<typename T>
  static void emit (EventGroup group, T event)

  Emits an event.

7.9.1 Detailed Description

Provides a simple interface for emitting events.

This class provides a static method for emitting events, which logs the event to the EventManager.

Definition at line 266 of file event_manager.h.

7.9.2 Member Function Documentation

7.9.2.1 emit()

```
template<typename T>
static void EventEmitter::emit (
            EventGroup group,
            T event)  [inline], [static]
```

Emits an event.

Parameters

in	*group*	Event group.
in	*event*	Event code.

Template Parameters

T	Type of the event enumeration.

Definition at line 275 of file event_manager.h.

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

- lib/eventman/event_manager.h

7.10 EventLog Class Reference

Structure for storing event log data.

```
#include <event_manager.h>
```

Public Attributes

- uint16_t id
  Unique identifier for the event.
- uint32_t timestamp
  Timestamp of the event in milliseconds since boot.
- uint8_t group
  Event group.
- uint8_t event
  Event code.

7.10.1 Detailed Description

Structure for storing event log data.

Represents a single event log entry with an ID, timestamp, group, and event code.

Definition at line 169 of file event_manager.h.

7.10.2 Member Data Documentation

7.10.2.1 id

```
uint16_t EventLog::id
```

Unique identifier for the event.

Definition at line 172 of file event_manager.h.

7.10.2.2 timestamp

```
uint32_t EventLog::timestamp
```

Timestamp of the event in milliseconds since boot.

Definition at line 174 of file event_manager.h.

7.10.2.3 group

```
uint8_t EventLog::group
```

Event group.

Definition at line 176 of file event_manager.h.

7.10.2.4 event

```
uint8_t EventLog::event
```

Event code.

Definition at line 178 of file event_manager.h.

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

- lib/eventman/event_manager.h

7.11 EventManager Class Reference

Manages event logging and storage.

```
#include <event_manager.h>
```

Public Member Functions

- bool init ()
  Initializes the event manager.
- void log_event (uint8_t group, uint8_t event)
  Logs an event to the event buffer.
- const EventLog & get_event (size_t index) const
  Gets an event from the event buffer.
- size_t get_event_count () const
  Gets the number of events in the buffer.
- bool save_to_storage ()
  Saves the event buffer to persistent storage.
- bool load_from_storage ()
  Loads the event buffer from persistent storage.

Static Public Member Functions

- static EventManager & get_instance ()
  Gets the singleton instance of the EventManager class.

Private Member Functions

- EventManager ()
- EventManager (const EventManager &)=delete
- EventManager & operator= (const EventManager &)=delete

Private Attributes

- EventLog events [EVENT_BUFFER_SIZE]
- size_t eventCount
- size_t writeIndex
- mutex_t eventMutex
- uint16_t nextEventId
- size_t eventsSinceFlush

7.11.1 Detailed Description

Manages event logging and storage.

This class provides a singleton instance for logging events to a circular buffer and saving them to persistent storage. It ensures thread-safe access to the event log and provides methods for initializing, logging, retrieving, saving, and loading events.

Definition at line 190 of file event_manager.h.

7.11.2 Constructor & Destructor Documentation

7.11.2.1 EventManager() [1/2]

```
EventManager::EventManager ()  [inline], [private]
```

Definition at line 199 of file event_manager.h.

7.11.2.2 EventManager() [2/2]

```
EventManager::EventManager (
            const EventManager & )  [private], [delete]
```

7.11.3 Member Function Documentation

7.11.3.1 operator=()

```
EventManager & EventManager::operator= (
            const EventManager & )  [private], [delete]
```

7.11.3.2 get_instance()

```
static EventManager & EventManager::get_instance ()  [inline], [static]
```

Gets the singleton instance of the EventManager class.

Returns

A reference to the singleton instance.

Definition at line 216 of file event_manager.h.

7.11.3.3 get_event_count()

```
size_t EventManager::get_event_count () const [inline]
```

Gets the number of events in the buffer.

Returns

The number of events in the buffer.

Definition at line 245 of file event_manager.h.

7.11.3.4 load_from_storage()

```
bool EventManager::load_from_storage ()
```

Loads the event buffer from persistent storage.

Returns

True if the load was successful, false otherwise.

7.11.4 Member Data Documentation

7.11.4.1 events

```
EventLog EventManager::events[EVENT_BUFFER_SIZE] [private]
```

Definition at line 192 of file event_manager.h.

7.11.4.2 eventCount

```
size_t EventManager::eventCount [private]
```

Definition at line 193 of file event_manager.h.

7.11.4.3 writeIndex

```
size_t EventManager::writeIndex [private]
```

Definition at line 194 of file event_manager.h.

7.11.4.4 eventMutex

```
mutex_t EventManager::eventMutex [private]
```

Definition at line 195 of file event_manager.h.

7.11.4.5 nextEventId

`uint16_t EventManager::nextEventId [private]`

Definition at line 196 of file event_manager.h.

7.11.4.6 eventsSinceFlush

`size_t EventManager::eventsSinceFlush [private]`

Definition at line 197 of file event_manager.h.

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

- lib/eventman/event_manager.h
- lib/eventman/event_manager.cpp

7.12 Frame Struct Reference

Represents a communication frame used for data exchange.

`#include <protocol.h>`

Public Attributes

- std::string header
- uint8_t direction
- OperationType operationType
- uint8_t group
- uint8_t command
- std::string value
- std::string unit
- std::string footer

7.12.1 Detailed Description

Represents a communication frame used for data exchange.

This structure encapsulates the different components of a communication frame, including the header, direction, operation type, group ID, command ID, payload value, unit, and footer. It is used for both encoding and decoding messages.

Note

The `header` and `footer` fields are used to mark the beginning and end of the frame, respectively.

The `direction` field indicates the direction of the communication (0 = ground->sat, 1 = sat->ground).

The `operationType` field specifies the type of operation being performed (e.g., GET, SET, ANS, ERR, INF).

The `group` and `command` fields identify the specific command being executed.

The `value` field contains the payload data.

The `unit` field specifies the unit of measurement for the payload data.

Definition at line 181 of file protocol.h.

7.12.2 Member Data Documentation

7.12.2.1 header

`std::string Frame::header`

Definition at line 182 of file protocol.h.

7.12.2.2 direction

`uint8_t Frame::direction`

Definition at line 183 of file protocol.h.

7.12.2.3 operationType

`OperationType Frame::operationType`

Definition at line 184 of file protocol.h.

7.12.2.4 group

`uint8_t Frame::group`

Definition at line 185 of file protocol.h.

7.12.2.5 command

`uint8_t Frame::command`

Definition at line 186 of file protocol.h.

7.12.2.6 value

`std::string Frame::value`

Definition at line 187 of file protocol.h.

7.12.2.7 unit

`std::string Frame::unit`

Definition at line 188 of file protocol.h.

7.12.2.8 footer

```
std::string Frame::footer
```

Definition at line 189 of file protocol.h.

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

- lib/comms/protocol.h

7.13 INA3221 Class Reference

INA3221 Triple-Channel Power Monitor driver class.

```
#include <INA3221.h>
```

Classes

- struct conf_reg_t

    Configuration register bit fields.
- struct masken_reg_t

    Mask/Enable register bit fields.

Public Member Functions

- INA3221 (ina3221_addr_t addr, i2c_inst_t ∗i2c)

    Constructor for INA3221 class.
- bool begin ()

    Initialize the INA3221 device.
- uint16_t read_register (ina3221_reg_t reg)

    Read a register from the device.
- void reset ()

    Reset the INA3221 to default settings.
- void set_mode_power_down ()

    Set device to power-down mode.
- void set_mode_continuous ()

    Set device to continuous measurement mode.
- void set_mode_triggered ()

    Set device to triggered measurement mode.
- void set_shunt_measurement_enable ()

    Enable shunt voltage measurements.
- void set_shunt_measurement_disable ()

    Disable shunt voltage measurements.
- void set_bus_measurement_enable ()

    Enable bus voltage measurements.
- void set_bus_measurement_disable ()

    Disable bus voltage measurements.
- void set_averaging_mode (ina3221_avg_mode_t mode)

Set the averaging mode for measurements.
- void set_bus_conversion_time (ina3221_conv_time_t convTime)

  Set bus voltage conversion time.
- void set_shunt_conversion_time (ina3221_conv_time_t convTime)

  Set shunt voltage conversion time.
- uint16_t get_manufacturer_id ()

  Get the manufacturer ID of the device.
- uint16_t get_die_id ()

  Get the die ID of the device.
- int32_t get_shunt_voltage (ina3221_ch_t channel)

  Get shunt voltage for a specific channel.
- float get_current (ina3221_ch_t channel)
- float get_current_ma (ina3221_ch_t channel)

  Get current for a specific channel.
- float get_voltage (ina3221_ch_t channel)

  Get bus voltage for a specific channel.

Private Member Functions

- void _read (ina3221_reg_t reg, uint16_t ∗val)

  Read a 16-bit register from the device.
- void _write (ina3221_reg_t reg, uint16_t ∗val)

  Write a 16-bit value to a register.

Private Attributes

- ina3221_addr_t _i2c_addr
- i2c_inst_t ∗ _i2c
- uint32_t _shuntRes [INA3221_CH_NUM]
- uint32_t _filterRes [INA3221_CH_NUM]
- masken_reg_t _masken_reg

7.13.1 Detailed Description

INA3221 Triple-Channel Power Monitor driver class.

Provides functionality for voltage, current, and power monitoring with configurable alerts and power valid monitoring

Definition at line 96 of file INA3221.h.

7.13.2 Member Function Documentation

7.13.2.1 _read()

```
void INA3221::_read (
            ina3221_reg_t reg,
            uint16_t ∗ val)  [private]
```

Read a 16-bit register from the device.

Parameters

reg	Register address
val	Pointer to store the read value

Definition at line 354 of file INA3221.cpp.

7.13.2.2 _write()

```
void INA3221::_write (
            ina3221_reg_t reg,
            uint16_t * val)  [private]
```

Write a 16-bit value to a register.

Parameters

reg	Register address
val	Pointer to the value to write

Definition at line 380 of file INA3221.cpp.

7.13.2.3 get_current()

```
float INA3221::get_current (
            ina3221_ch_t channel)
```

7.13.3 Member Data Documentation

7.13.3.1 _i2c_addr

```
ina3221_addr_t INA3221::_i2c_addr  [private]
```

Definition at line 137 of file INA3221.h.

7.13.3.2 _i2c

```
i2c_inst_t* INA3221::_i2c  [private]
```

Definition at line 138 of file INA3221.h.

7.13.3.3 _shuntRes

```
uint32_t INA3221::_shuntRes[INA3221_CH_NUM]  [private]
```

Definition at line 141 of file INA3221.h.

7.13.3.4 _filterRes

```
uint32_t INA3221::_filterRes[INA3221_CH_NUM] [private]
```

Definition at line 144 of file INA3221.h.

7.13.3.5 _masken_reg

```
masken_reg_t INA3221::_masken_reg [private]
```

Definition at line 147 of file INA3221.h.

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

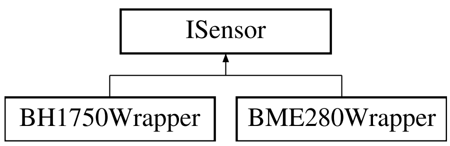
- lib/powerman/INA3221/INA3221.h
- lib/powerman/INA3221/INA3221.cpp

7.14 ISensor Class Reference

Abstract base class for sensors.

```
#include <ISensor.h>
```

Inheritance diagram for ISensor:



Public Member Functions

- virtual ∼ISensor ()=default

  Virtual destructor.
- virtual bool init ()=0

  Initializes the sensor.
- virtual float read_data (SensorDataTypeIdentifier type)=0

  Reads data from the sensor.
- virtual bool is_initialized () const =0

  Checks if the sensor is initialized.
- virtual SensorType get_type () const =0

  Gets the sensor type.
- virtual bool configure (const std::map< std::string, std::string > &config)=0

  Configures the sensor.
- virtual uint8_t get_address () const =0

  Gets the I2C address of the sensor.

7.14.1 Detailed Description

Abstract base class for sensors.

Defines the interface for interacting with different types of sensors.

Definition at line 63 of file ISensor.h.

7.14.2 Constructor & Destructor Documentation

7.14.2.1 ~ISensor()

```
virtual ISensor::~ISensor ()  [virtual], [default]
```

Virtual destructor.

Ensures proper cleanup of derived classes.

7.14.3 Member Function Documentation

7.14.3.1 init()

```
virtual bool ISensor::init ()  [pure virtual]
```

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implemented in BH1750Wrapper, and BME280Wrapper.

7.14.3.2 read_data()

```
virtual float ISensor::read_data (
            SensorDataTypeIdentifier type)  [pure virtual]
```

Reads data from the sensor.

Parameters

in	*type*	Data type to read.

Returns

The sensor data.

Implemented in BH1750Wrapper, and BME280Wrapper.

7.14.3.3 is_initialized()

```
virtual bool ISensor::is_initialized () const [pure virtual]
```

Checks if the sensor is initialized.

Returns

True if the sensor is initialized, false otherwise.

Implemented in BH1750Wrapper, and BME280Wrapper.

7.14.3.4 get_type()

```
virtual SensorType ISensor::get_type () const [pure virtual]
```

Gets the sensor type.

Returns

The sensor type.

Implemented in BH1750Wrapper, and BME280Wrapper.

7.14.3.5 configure()

```
virtual bool ISensor::configure (
            const std::map< std::string, std::string > & config) [pure virtual]
```

Configures the sensor.

Parameters

in	*config*	A map of configuration parameters.

Returns

True if configuration was successful, false otherwise.

Implemented in BH1750Wrapper, and BME280Wrapper.

7.14.3.6 get_address()

```
virtual uint8_t ISensor::get_address () const [pure virtual]
```

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

Implemented in BH1750Wrapper, and BME280Wrapper.

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

- lib/sensors/ISensor.h

7.15 INA3221::masken_reg_t Struct Reference

Mask/Enable register bit fields.

Public Attributes

- uint16_t conv_ready:1
- uint16_t timing_ctrl_alert:1
- uint16_t pwr_valid_alert:1
- uint16_t warn_alert_ch3:1
- uint16_t warn_alert_ch2:1
- uint16_t warn_alert_ch1:1
- uint16_t shunt_sum_alert:1
- uint16_t crit_alert_ch3:1
- uint16_t crit_alert_ch2:1
- uint16_t crit_alert_ch1:1
- uint16_t crit_alert_latch_en:1
- uint16_t warn_alert_latch_en:1
- uint16_t shunt_sum_en_ch3:1
- uint16_t shunt_sum_en_ch2:1
- uint16_t shunt_sum_en_ch1:1
- uint16_t reserved:1

7.15.1 Detailed Description

Mask/Enable register bit fields.

Definition at line 117 of file INA3221.h.

7.15.2 Member Data Documentation

7.15.2.1 conv_ready

```
uint16_t INA3221::masken_reg_t::conv_ready
```

Definition at line 118 of file INA3221.h.

7.15.2.2 timing_ctrl_alert

```
uint16_t INA3221::masken_reg_t::timing_ctrl_alert
```

Definition at line 119 of file INA3221.h.

7.15.2.3 pwr_valid_alert

```
uint16_t INA3221::masken_reg_t::pwr_valid_alert
```

Definition at line 120 of file INA3221.h.

7.15.2.4 warn_alert_ch3

`uint16_t INA3221::masken_reg_t::warn_alert_ch3`

Definition at line 121 of file INA3221.h.

7.15.2.5 warn_alert_ch2

`uint16_t INA3221::masken_reg_t::warn_alert_ch2`

Definition at line 122 of file INA3221.h.

7.15.2.6 warn_alert_ch1

`uint16_t INA3221::masken_reg_t::warn_alert_ch1`

Definition at line 123 of file INA3221.h.

7.15.2.7 shunt_sum_alert

`uint16_t INA3221::masken_reg_t::shunt_sum_alert`

Definition at line 124 of file INA3221.h.

7.15.2.8 crit_alert_ch3

`uint16_t INA3221::masken_reg_t::crit_alert_ch3`

Definition at line 125 of file INA3221.h.

7.15.2.9 crit_alert_ch2

`uint16_t INA3221::masken_reg_t::crit_alert_ch2`

Definition at line 126 of file INA3221.h.

7.15.2.10 crit_alert_ch1

`uint16_t INA3221::masken_reg_t::crit_alert_ch1`

Definition at line 127 of file INA3221.h.

7.15.2.11 crit_alert_latch_en

`uint16_t INA3221::masken_reg_t::crit_alert_latch_en`

Definition at line 128 of file INA3221.h.

7.15.2.12 warn_alert_latch_en

`uint16_t INA3221::masken_reg_t::warn_alert_latch_en`

Definition at line 129 of file INA3221.h.

7.15.2.13 shunt_sum_en_ch3

`uint16_t INA3221::masken_reg_t::shunt_sum_en_ch3`

Definition at line 130 of file INA3221.h.

7.15.2.14 shunt_sum_en_ch2

`uint16_t INA3221::masken_reg_t::shunt_sum_en_ch2`

Definition at line 131 of file INA3221.h.

7.15.2.15 shunt_sum_en_ch1

`uint16_t INA3221::masken_reg_t::shunt_sum_en_ch1`

Definition at line 132 of file INA3221.h.

7.15.2.16 reserved

`uint16_t INA3221::masken_reg_t::reserved`

Definition at line 133 of file INA3221.h.

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

- lib/powerman/INA3221/INA3221.h

7.16 NMEAData Class Reference

Manages parsed NMEA sentences.

`#include <NMEA_data.h>`

Public Member Functions

- void update_rmc_tokens (const std::vector< std::string > &tokens)

  Updates the RMC tokens with new data.
- void update_gga_tokens (const std::vector< std::string > &tokens)

  Updates the GGA tokens with new data.
- std::vector< std::string > get_rmc_tokens () const

  Gets a copy of the RMC tokens.
- std::vector< std::string > get_gga_tokens () const

  Gets a copy of the GGA tokens.
- bool has_valid_time () const

  Checks if the NMEA data has valid time information.
- time_t get_unix_time () const

  Converts the NMEA data to a Unix timestamp.

Static Public Member Functions

- static NMEAData & get_instance ()

  Gets the singleton instance of the NMEAData class.

Private Member Functions

- NMEAData ()

  Private constructor for the singleton pattern.
- NMEAData (const NMEAData &)=delete

  Deleted copy constructor to prevent copying.
- NMEAData & operator= (const NMEAData &)=delete

  Deleted assignment operator to prevent assignment.

Private Attributes

- std::vector< std::string > rmc_tokens_

  Vector of tokens from the most recent RMC sentence.
- std::vector< std::string > gga_tokens_

  Vector of tokens from the most recent GGA sentence.
- mutex_t rmc_mutex_

  Mutex for thread-safe access to the RMC tokens.
- mutex_t gga_mutex_

  Mutex for thread-safe access to the GGA tokens.

7.16.1 Detailed Description

Manages parsed NMEA sentences.

This class is a singleton that stores and provides access to parsed data from NMEA sentences received from a GPS module. It includes methods for updating and retrieving RMC and GGA tokens, as well as converting the data to a Unix timestamp.

Definition at line 33 of file NMEA_data.h.

7.16.2 Constructor & Destructor Documentation

7.16.2.1 NMEAData() [1/2]

```
NMEAData::NMEAData ()  [inline], [private]
```

Private constructor for the singleton pattern.

Initializes the mutexes.

Definition at line 48 of file NMEA_data.h.

7.16.2.2 NMEAData() [2/2]

```
NMEAData::NMEAData (
            const NMEAData & )  [private], [delete]
```

Deleted copy constructor to prevent copying.

7.16.3 Member Function Documentation

7.16.3.1 operator=()

```
NMEAData & NMEAData::operator= (
            const NMEAData & )  [private], [delete]
```

Deleted assignment operator to prevent assignment.

7.16.3.2 get_instance()

```
static NMEAData & NMEAData::get_instance ()  [inline], [static]
```

Gets the singleton instance of the NMEAData class.

Returns

A reference to the singleton instance.

Definition at line 67 of file NMEA_data.h.

7.16.3.3 update_rmc_tokens()

```
void NMEAData::update_rmc_tokens (
            const std::vector< std::string > & tokens)  [inline]
```

Updates the RMC tokens with new data.

Parameters

in	*tokens*	Vector of strings representing the RMC tokens.

Definition at line 76 of file NMEA_data.h.

7.16.3.4 update_gga_tokens()

```
void NMEAData::update_gga_tokens (
            const std::vector< std::string > & tokens)  [inline]
```

Updates the GGA tokens with new data.

Parameters

in	*tokens*	Vector of strings representing the GGA tokens.

Definition at line 86 of file NMEA_data.h.

7.16.3.5 get_rmc_tokens()

```
std::vector< std::string > NMEAData::get_rmc_tokens () const  [inline]
```

Gets a copy of the RMC tokens.

Returns

A copy of the RMC tokens.

Definition at line 96 of file NMEA_data.h.

7.16.3.6 get_gga_tokens()

```
std::vector< std::string > NMEAData::get_gga_tokens () const  [inline]
```

Gets a copy of the GGA tokens.

Returns

A copy of the GGA tokens.

Definition at line 107 of file NMEA_data.h.

7.16.3.7 has_valid_time()

```
bool NMEAData::has_valid_time () const  [inline]
```

Checks if the NMEA data has valid time information.

Returns

True if the data has valid time information, false otherwise.

Definition at line 118 of file NMEA_data.h.

7.16.3.8 get_unix_time()

```
time_t NMEAData::get_unix_time () const  [inline]
```

Converts the NMEA data to a Unix timestamp.

Returns

The Unix timestamp, or 0 if the data is invalid.

Definition at line 126 of file NMEA_data.h.

7.16.4 Member Data Documentation

7.16.4.1 rmc_tokens_

```
std::vector<std::string> NMEAData::rmc_tokens_  [private]
```

Vector of tokens from the most recent RMC sentence.

Definition at line 36 of file NMEA_data.h.

7.16.4.2 gga_tokens_

```
std::vector<std::string> NMEAData::gga_tokens_  [private]
```

Vector of tokens from the most recent GGA sentence.

Definition at line 38 of file NMEA_data.h.

7.16.4.3 rmc_mutex_

```
mutex_t NMEAData::rmc_mutex_  [private]
```

Mutex for thread-safe access to the RMC tokens.

Definition at line 40 of file NMEA_data.h.

7.16.4.4 gga_mutex_

```
mutex_t NMEAData::gga_mutex_ [private]
```

Mutex for thread-safe access to the GGA tokens.

Definition at line 42 of file NMEA_data.h.

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

- lib/location/NMEA/NMEA_data.h

7.17 PowerManager Class Reference

Manages power-related functions.

```
#include <PowerManager.h>
```

Public Member Functions

- PowerManager (i2c_inst_t ∗i2c)

  Constructor for the PowerManager class.
- bool initialize ()

  Initializes the PowerManager.
- std::string read_device_ids ()

  Reads the manufacturer and die IDs from the INA3221.
- float get_current_charge_solar ()

  Gets the solar charging current.
- float get_current_charge_usb ()

  Gets the USB charging current.
- float get_current_charge_total ()

  Gets the total charging current.
- float get_current_draw ()

  Gets the current draw.
- float get_voltage_battery ()

  Gets the battery voltage.
- float get_voltage_5v ()

  Gets the 5V voltage.
- void configure (const std::map< std::string, std::string > &config)

  Configures the INA3221.
- bool is_charging_solar ()

  Checks if solar charging is active.
- bool is_charging_usb ()

  Checks if USB charging is active.

Static Public Member Functions

- static PowerManager & get_instance ()

  Gets the singleton instance of the PowerManager class.

Static Public Attributes

- static constexpr float SOLAR_CURRENT_THRESHOLD = 50.0f

  Solar current threshold in milliamperes.
- static constexpr float USB_CURRENT_THRESHOLD = 50.0f

  USB current threshold in milliamperes.
- static constexpr float BATTERY_LOW_THRESHOLD = 2.8f

  Low voltage threshold in volts.
- static constexpr float BATTERY_FULL_THRESHOLD = 4.2f

  Overcharge voltage threshold in volts.

Private Member Functions

- PowerManager ()

  Private constructor for the singleton pattern.
- PowerManager (const PowerManager &)=delete

  Deleted copy constructor to prevent copying.
- PowerManager & operator= (const PowerManager &)=delete

  Deleted assignment operator to prevent assignment.

Private Attributes

- INA3221 ina3221_

  INA3221 instance for power monitoring.
- bool initialized_

  Flag indicating if the PowerManager is initialized.
- recursive_mutex_t powerman_mutex_

  Mutex for thread-safe access to the PowerManager.
- bool charging_solar_active_ = false

  Flag indicating if solar charging is active.
- bool charging_usb_active_ = false

  Flag indicating if USB charging is active.

7.17.1 Detailed Description

Manages power-related functions.

This class is a singleton that provides methods for reading voltage and current values, configuring the INA3221 power monitor, and checking power alerts.

Definition at line 32 of file PowerManager.h.

7.17.2 Constructor & Destructor Documentation

7.17.2.1 PowerManager() [1/2]

```
PowerManager::PowerManager (
            i2c_inst_t * i2c)
```

Constructor for the PowerManager class.

Parameters

in	*i2c*	I2C instance to use for communication with the INA3221.

7.17.2.2 PowerManager() [2/2]

```
PowerManager::PowerManager (
            const PowerManager & )  [private], [delete]
```

Deleted copy constructor to prevent copying.

7.17.3 Member Function Documentation

7.17.3.1 operator=()

```
PowerManager & PowerManager::operator= (
            const PowerManager & )  [private], [delete]
```

Deleted assignment operator to prevent assignment.

7.17.4 Member Data Documentation

7.17.4.1 SOLAR_CURRENT_THRESHOLD

```
float PowerManager::SOLAR_CURRENT_THRESHOLD = 50.0f  [static], [constexpr]
```

Solar current threshold in milliamperes.

Definition at line 114 of file PowerManager.h.

7.17.4.2 USB_CURRENT_THRESHOLD

```
float PowerManager::USB_CURRENT_THRESHOLD = 50.0f  [static], [constexpr]
```

USB current threshold in milliamperes.

Definition at line 116 of file PowerManager.h.

7.17.4.3 BATTERY_LOW_THRESHOLD

```
float PowerManager::BATTERY_LOW_THRESHOLD = 2.8f  [static], [constexpr]
```

Low voltage threshold in volts.

Definition at line 118 of file PowerManager.h.

7.17.4.4 BATTERY_FULL_THRESHOLD

`float PowerManager::BATTERY_FULL_THRESHOLD = 4.2f [static], [constexpr]`

Overcharge voltage threshold in volts.

Definition at line 120 of file PowerManager.h.

7.17.4.5 ina3221_

`INA3221 PowerManager::ina3221_ [private]`

INA3221 instance for power monitoring.

Definition at line 124 of file PowerManager.h.

7.17.4.6 initialized_

`bool PowerManager::initialized_ [private]`

Flag indicating if the PowerManager is initialized.

Definition at line 126 of file PowerManager.h.

7.17.4.7 powerman_mutex_

`recursive_mutex_t PowerManager::powerman_mutex_ [private]`

Mutex for thread-safe access to the PowerManager.

Definition at line 128 of file PowerManager.h.

7.17.4.8 charging_solar_active_

`bool PowerManager::charging_solar_active_ = false [private]`

Flag indicating if solar charging is active.

Definition at line 130 of file PowerManager.h.

7.17.4.9 charging_usb_active_

`bool PowerManager::charging_usb_active_ = false [private]`

Flag indicating if USB charging is active.

Definition at line 132 of file PowerManager.h.

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

- lib/powerman/PowerManager.h
- lib/powerman/PowerManager.cpp

7.18 SensorDataRecord Struct Reference

Structure representing a single sensor data point.

```
#include <telemetry_manager.h>
```

Public Member Functions

- std::string to_csv () const
  Converts the sensor data record to a CSV string.

Public Attributes

- uint32_t timestamp
- float temperature
- float pressure
- float humidity
- float light

7.18.1 Detailed Description

Structure representing a single sensor data point.

Contains measurements from the environment and light sensors collected at a specific point in time

Definition at line 110 of file telemetry_manager.h.

7.18.2 Member Data Documentation

7.18.2.1 timestamp

```
uint32_t SensorDataRecord::timestamp
```

Unix timestamp of the record

Definition at line 111 of file telemetry_manager.h.

7.18.2.2 temperature

```
float SensorDataRecord::temperature
```

Temperature in degrees Celsius

Definition at line 112 of file telemetry_manager.h.

7.18.2.3 pressure

`float SensorDataRecord::pressure`

Pressure in hPa

Definition at line 113 of file telemetry_manager.h.

7.18.2.4 humidity

`float SensorDataRecord::humidity`

Relative humidity in %

Definition at line 114 of file telemetry_manager.h.

7.18.2.5 light

`float SensorDataRecord::light`

Light intensity in lux

Definition at line 115 of file telemetry_manager.h.

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

- lib/telemetry/telemetry_manager.h

7.19 SensorWrapper Class Reference

Manages a collection of sensors.

`#include <ISensor.h>`

Public Member Functions

- bool sensor_init (SensorType type, i2c_inst_t *i2c=nullptr)
  Initializes a sensor.
- bool sensor_configure (SensorType type, const std::map< std::string, std::string > &config)
  Configures a sensor.
- float sensor_read_data (SensorType sensorType, SensorDataTypeIdentifier dataType)
  Reads data from a sensor.
- ISensor * get_sensor (SensorType type)
  Gets a sensor.
- std::vector< std::pair< SensorType, uint8_t > > scan_connected_sensors (i2c_inst_t *i2c)
  Scans for connected sensors.
- std::vector< std::pair< SensorType, uint8_t > > get_available_sensors ()
  Gets a list of available sensors.

Static Public Member Functions

- static SensorWrapper & get_instance ()
  Gets the singleton instance of the SensorWrapper class.

Private Member Functions

- SensorWrapper ()=default
  Private constructor for the singleton pattern.

Private Attributes

- std::map$<$ SensorType, ISensor $*$ $>$ sensors
  Map of sensor types to sensor instances.

7.19.1 Detailed Description

Manages a collection of sensors.

This class provides methods for initializing, configuring, and reading data from different types of sensors.

Definition at line 116 of file ISensor.h.

7.19.2 Constructor & Destructor Documentation

7.19.2.1 SensorWrapper()

```
SensorWrapper::SensorWrapper ()  [private], [default]
```

Private constructor for the singleton pattern.

7.19.3 Member Function Documentation

7.19.3.1 get_instance()

```
static SensorWrapper & SensorWrapper::get_instance ()  [inline], [static]
```

Gets the singleton instance of the SensorWrapper class.

Returns

A reference to the singleton instance.

Definition at line 122 of file ISensor.h.

7.19.4 Member Data Documentation

7.19.4.1 sensors

`std::map<SensorType, ISensor*> SensorWrapper::sensors [private]`

Map of sensor types to sensor instances.

Definition at line 173 of file ISensor.h.

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

- lib/sensors/ISensor.h
- lib/sensors/ISensor.cpp

7.20 SystemStateManager Class Reference

Manages the system state of the Kubisat firmware.

`#include <system_state_manager.h>`

Public Member Functions

- bool is_bootloader_reset_pending () const

  Checks if a bootloader reset is pending.
- void set_bootloader_reset_pending (bool pending)

  Sets whether a bootloader reset is pending.
- bool is_gps_collection_paused () const

  Checks if GPS collection is paused.
- void set_gps_collection_paused (bool paused)

  Sets whether GPS collection is paused.
- bool is_sd_card_mounted () const

  Checks if the SD card is mounted.
- void set_sd_card_mounted (bool mounted)

  Sets whether the SD card is mounted.
- VerbosityLevel get_uart_verbosity () const

  Gets the UART verbosity level.
- void set_uart_verbosity (VerbosityLevel level)

  Sets the UART verbosity level.
- bool is_radio_init_ok () const

  Checks if the radio initialization was successful.
- void set_radio_init_ok (bool status)

  Sets whether the radio initialization was successful.
- bool is_light_sensor_init_ok () const

  Checks if the light sensor initialization was successful.
- void set_light_sensor_init_ok (bool status)

  Sets whether the light sensor initialization was successful.
- bool is_env_sensor_init_ok () const

  Checks if the environment sensor initialization was successful.
- void set_env_sensor_init_ok (bool status)

  Sets whether the environment sensor initialization was successful.

Static Public Member Functions

- static SystemStateManager & get_instance ()

  Gets the singleton instance of the SystemStateManager class.

Private Member Functions

- SystemStateManager ()

  Private constructor for the singleton pattern.
- SystemStateManager (const SystemStateManager &)=delete

  Deleted copy constructor to prevent copying.
- SystemStateManager & operator= (const SystemStateManager &)=delete

  Deleted assignment operator to prevent assignment.

Private Attributes

- bool pending_bootloader_reset

  Flag indicating whether a bootloader reset is pending.
- bool gps_collection_paused

  Flag indicating whether GPS collection is paused.
- bool sd_card_mounted

  Flag indicating whether the SD card is mounted.
- VerbosityLevel uart_verbosity

  The UART verbosity level.
- bool sd_card_init_status

  Flag indicating whether the SD card initialization was successful.
- bool radio_init_status

  Flag indicating whether the radio initialization was successful.
- bool light_sensor_init_status

  Flag indicating whether the light sensor initialization was successful.
- bool env_sensor_init_status

  Flag indicating whether the environment sensor initialization was successful.
- recursive_mutex_t mutex_

  Mutex for thread-safe access to the system state.

7.20.1 Detailed Description

Manages the system state of the Kubisat firmware.

This class is a singleton that provides methods for getting and setting various system states, such as whether a bootloader reset is pending, whether GPS collection is paused, whether the SD card is mounted, and the UART verbosity level.

Definition at line 32 of file system_state_manager.h.

7.20.2 Constructor & Destructor Documentation

7.20.2.1 SystemStateManager() [1/2]

```
SystemStateManager::SystemStateManager ()  [inline], [private]
```

Private constructor for the singleton pattern.

Initializes the system state and mutex.

Definition at line 57 of file system_state_manager.h.

7.20.2.2 SystemStateManager() [2/2]

```
SystemStateManager::SystemStateManager (
            const SystemStateManager & )  [private], [delete]
```

Deleted copy constructor to prevent copying.

7.20.3 Member Function Documentation

7.20.3.1 operator=()

```
SystemStateManager & SystemStateManager::operator= (
            const SystemStateManager & )  [private], [delete]
```

Deleted assignment operator to prevent assignment.

7.20.3.2 get_instance()

```
static SystemStateManager & SystemStateManager::get_instance ()  [inline], [static]
```

Gets the singleton instance of the SystemStateManager class.

Returns

A reference to the singleton instance.

Definition at line 84 of file system_state_manager.h.

7.20.3.3 is_bootloader_reset_pending()

```
bool SystemStateManager::is_bootloader_reset_pending () const  [inline]
```

Checks if a bootloader reset is pending.

Returns

True if a bootloader reset is pending, false otherwise.

Definition at line 93 of file system_state_manager.h.

7.20.3.4 set_bootloader_reset_pending()

```
void SystemStateManager::set_bootloader_reset_pending (
            bool pending)  [inline]
```

Sets whether a bootloader reset is pending.

Parameters

in	*pending*	True if a bootloader reset is pending, false otherwise.

Definition at line 104 of file system_state_manager.h.

7.20.3.5 is_gps_collection_paused()

```
bool SystemStateManager::is_gps_collection_paused ( ) const [inline]
```

Checks if GPS collection is paused.

Returns

True if GPS collection is paused, false otherwise.

Definition at line 114 of file system_state_manager.h.

7.20.3.6 set_gps_collection_paused()

```
void SystemStateManager::set_gps_collection_paused (
            bool paused ) [inline]
```

Sets whether GPS collection is paused.

Parameters

in	*paused*	True if GPS collection is paused, false otherwise.

Definition at line 125 of file system_state_manager.h.

7.20.3.7 is_sd_card_mounted()

```
bool SystemStateManager::is_sd_card_mounted ( ) const [inline]
```

Checks if the SD card is mounted.

Returns

True if the SD card is mounted, false otherwise.

Definition at line 135 of file system_state_manager.h.

7.20.3.8 set_sd_card_mounted()

```
void SystemStateManager::set_sd_card_mounted (
            bool mounted ) [inline]
```

Sets whether the SD card is mounted.

Parameters

in	*mounted*	True if the SD card is mounted, false otherwise.

Definition at line 146 of file system_state_manager.h.

7.20.3.9 get_uart_verbosity()

```
VerbosityLevel SystemStateManager::get_uart_verbosity () const  [inline]
```

Gets the UART verbosity level.

Returns

The UART verbosity level.

Definition at line 156 of file system_state_manager.h.

7.20.3.10 set_uart_verbosity()

```
void SystemStateManager::set_uart_verbosity (
            VerbosityLevel level)  [inline]
```

Sets the UART verbosity level.

Parameters

in	*level*	The UART verbosity level.

Definition at line 167 of file system_state_manager.h.

7.20.3.11 is_radio_init_ok()

```
bool SystemStateManager::is_radio_init_ok () const  [inline]
```

Checks if the radio initialization was successful.

Returns

True if the radio initialization was successful, false otherwise.

Definition at line 177 of file system_state_manager.h.

7.20.3.12 set_radio_init_ok()

```
void SystemStateManager::set_radio_init_ok (
            bool status)  [inline]
```

Sets whether the radio initialization was successful.

Parameters

in	*status*	True if the radio initialization was successful, false otherwise.

Definition at line 188 of file system_state_manager.h.

7.20.3.13 is_light_sensor_init_ok()

```
bool SystemStateManager::is_light_sensor_init_ok () const  [inline]
```

Checks if the light sensor initialization was successful.

Returns

True if the light sensor initialization was successful, false otherwise.

Definition at line 198 of file system_state_manager.h.

7.20.3.14 set_light_sensor_init_ok()

```
void SystemStateManager::set_light_sensor_init_ok (
            bool status)  [inline]
```

Sets whether the light sensor initialization was successful.

Parameters

in	*status*	True if the light sensor initialization was successful, false otherwise.

Definition at line 209 of file system_state_manager.h.

7.20.3.15 is_env_sensor_init_ok()

```
bool SystemStateManager::is_env_sensor_init_ok () const  [inline]
```

Checks if the environment sensor initialization was successful.

Returns

True if the environment sensor initialization was successful, false otherwise.

Definition at line 219 of file system_state_manager.h.

7.20.3.16 set_env_sensor_init_ok()

```
void SystemStateManager::set_env_sensor_init_ok (
            bool status)  [inline]
```

Sets whether the environment sensor initialization was successful.

Parameters

in	*status*	True if the environment sensor initialization was successful, false otherwise.

Definition at line 230 of file system_state_manager.h.

7.20.4 Member Data Documentation

7.20.4.1 pending_bootloader_reset

`bool SystemStateManager::pending_bootloader_reset [private]`

Flag indicating whether a bootloader reset is pending.

Definition at line 35 of file system_state_manager.h.

7.20.4.2 gps_collection_paused

`bool SystemStateManager::gps_collection_paused [private]`

Flag indicating whether GPS collection is paused.

Definition at line 37 of file system_state_manager.h.

7.20.4.3 sd_card_mounted

`bool SystemStateManager::sd_card_mounted [private]`

Flag indicating whether the SD card is mounted.

Definition at line 39 of file system_state_manager.h.

7.20.4.4 uart_verbosity

`VerbosityLevel SystemStateManager::uart_verbosity [private]`

The UART verbosity level.

Definition at line 41 of file system_state_manager.h.

7.20.4.5 sd_card_init_status

`bool SystemStateManager::sd_card_init_status [private]`

Flag indicating whether the SD card initialization was successful.

Definition at line 43 of file system_state_manager.h.

7.20.4.6 radio_init_status

```
bool SystemStateManager::radio_init_status [private]
```

Flag indicating whether the radio initialization was successful.

Definition at line 45 of file system_state_manager.h.

7.20.4.7 light_sensor_init_status

```
bool SystemStateManager::light_sensor_init_status [private]
```

Flag indicating whether the light sensor initialization was successful.

Definition at line 47 of file system_state_manager.h.

7.20.4.8 env_sensor_init_status

```
bool SystemStateManager::env_sensor_init_status [private]
```

Flag indicating whether the environment sensor initialization was successful.

Definition at line 49 of file system_state_manager.h.

7.20.4.9 mutex_

```
recursive_mutex_t SystemStateManager::mutex_ [private]
```

Mutex for thread-safe access to the system state.

Definition at line 51 of file system_state_manager.h.

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

- lib/system_state_manager.h

7.21 TelemetryManager Class Reference

Manages the collection, storage, and retrieval of telemetry data.

```
#include <telemetry_manager.h>
```

Public Member Functions

- bool init ()
  Initialize the telemetry system.
- bool collect_telemetry ()
  Collect telemetry data from sensors and power subsystems.
- void collect_power_telemetry (TelemetryRecord &record)
  Collects power subsystem telemetry data.
- void emit_power_events (float battery_voltage, float charge_current_usb, float charge_current_solar)
  Emits power-related events based on current and voltage levels.
- void collect_gps_telemetry (TelemetryRecord &record)
  Collects GPS telemetry data.
- void collect_sensor_telemetry (SensorDataRecord &sensor_record)
  Collects sensor telemetry data.
- bool flush_telemetry ()
  Save buffered telemetry data to storage.
- bool flush_sensor_data ()
  Save buffered sensor data to storage.
- bool is_telemetry_collection_time (uint32_t current_time, uint32_t &last_collection_time)
  Check if it's time to collect telemetry based on interval.
- bool is_telemetry_flush_time (uint32_t &collection_counter)
  Check if it's time to flush telemetry buffer based on count.
- std::string get_last_telemetry_record_csv ()
  Gets the last telemetry record as a CSV string.
- std::string get_last_sensor_record_csv ()
  Gets the last sensor data record as a CSV string.
- TelemetryRecord & get_last_telemetry_record ()
- SensorDataRecord & get_last_sensor_record ()
- size_t get_telemetry_buffer_count () const
- size_t get_telemetry_buffer_write_index () const

Static Public Member Functions

- static TelemetryManager & get_instance ()
  Gets the singleton instance of the TelemetryManager class.

Static Public Attributes

- static constexpr int TELEMETRY_BUFFER_SIZE = 20

Private Member Functions

- TelemetryManager ()
- ∼TelemetryManager ()=default

Private Attributes

- uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS
- uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD

  Current flush threshold (number of records that triggers a flush)
- TelemetryRecord telemetry_buffer [TELEMETRY_BUFFER_SIZE]

  Circular buffer for telemetry records.
- size_t telemetry_buffer_count = 0
- size_t telemetry_buffer_write_index = 0
- SensorDataRecord sensor_data_buffer [TELEMETRY_BUFFER_SIZE]

  Circular buffer for sensor data records.
- mutex_t telemetry_mutex

  Mutex for thread-safe access to the telemetry buffer.

Static Private Attributes

- static constexpr uint32_t DEFAULT_SAMPLE_INTERVAL_MS = 1000

  Current sampling interval in milliseconds.
- static constexpr uint32_t DEFAULT_FLUSH_THRESHOLD = 10

  Default number of records to collect before flushing to storage.

7.21.1 Detailed Description

Manages the collection, storage, and retrieval of telemetry data.

This class implements a singleton pattern to provide a single point of access for managing telemetry data. It handles the collection of data from various subsystems, stores the data in circular buffers, and provides methods for flushing the data to persistent storage and retrieving the last recorded data.

Definition at line 145 of file telemetry_manager.h.

7.21.2 Constructor & Destructor Documentation

7.21.2.1 ∼TelemetryManager()

```
TelemetryManager::∼TelemetryManager ()  [private], [default]
```

7.21.3 Member Function Documentation

7.21.3.1 get_instance()

```
static TelemetryManager & TelemetryManager::get_instance ()  [inline], [static]
```

Gets the singleton instance of the TelemetryManager class.

Returns

A reference to the singleton instance.

Definition at line 151 of file telemetry_manager.h.

7.21.3.2 flush_sensor_data()

```
bool TelemetryManager::flush_sensor_data ()
```

Save buffered sensor data to storage.

Returns

True if data was successfully saved

Writes all records from the sensor data buffer to the CSV file and clears the buffer after successful writing

7.21.3.3 get_last_telemetry_record()

```
TelemetryRecord & TelemetryManager::get_last_telemetry_record () [inline]
```

Definition at line 250 of file telemetry_manager.h.

7.21.3.4 get_last_sensor_record()

```
SensorDataRecord & TelemetryManager::get_last_sensor_record () [inline]
```

Definition at line 255 of file telemetry_manager.h.

7.21.3.5 get_telemetry_buffer_count()

```
size_t TelemetryManager::get_telemetry_buffer_count () const [inline]
```

Definition at line 260 of file telemetry_manager.h.

7.21.3.6 get_telemetry_buffer_write_index()

```
size_t TelemetryManager::get_telemetry_buffer_write_index () const [inline]
```

Definition at line 261 of file telemetry_manager.h.

7.21.4 Member Data Documentation

7.21.4.1 TELEMETRY_BUFFER_SIZE

```
int TelemetryManager::TELEMETRY_BUFFER_SIZE = 20 [static], [constexpr]
```

Definition at line 248 of file telemetry_manager.h.

7.21.4.2 DEFAULT_SAMPLE_INTERVAL_MS

```
uint32_t TelemetryManager::DEFAULT_SAMPLE_INTERVAL_MS = 1000  [static], [constexpr], [private]
```

Current sampling interval in milliseconds.

Definition at line 270 of file telemetry_manager.h.

7.21.4.3 DEFAULT_FLUSH_THRESHOLD

```
uint32_t TelemetryManager::DEFAULT_FLUSH_THRESHOLD = 10  [static], [constexpr], [private]
```

Default number of records to collect before flushing to storage.

Definition at line 275 of file telemetry_manager.h.

7.21.4.4 sample_interval_ms

```
uint32_t TelemetryManager::sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS  [private]
```

Definition at line 277 of file telemetry_manager.h.

7.21.4.5 flush_threshold

```
uint32_t TelemetryManager::flush_threshold = DEFAULT_FLUSH_THRESHOLD  [private]
```

Current flush threshold (number of records that triggers a flush)

Definition at line 282 of file telemetry_manager.h.

7.21.4.6 telemetry_buffer

```
TelemetryRecord TelemetryManager::telemetry_buffer[TELEMETRY_BUFFER_SIZE]  [private]
```

Circular buffer for telemetry records.

Definition at line 287 of file telemetry_manager.h.

7.21.4.7 telemetry_buffer_count

```
size_t TelemetryManager::telemetry_buffer_count = 0  [private]
```

Definition at line 288 of file telemetry_manager.h.

7.21.4.8 telemetry_buffer_write_index

```
size_t TelemetryManager::telemetry_buffer_write_index = 0  [private]
```

Definition at line 289 of file telemetry_manager.h.

7.21.4.9 sensor_data_buffer

```
SensorDataRecord TelemetryManager::sensor_data_buffer[TELEMETRY_BUFFER_SIZE]  [private]
```

Circular buffer for sensor data records.

Definition at line 294 of file telemetry_manager.h.

7.21.4.10 telemetry_mutex

```
mutex_t TelemetryManager::telemetry_mutex  [private]
```

Mutex for thread-safe access to the telemetry buffer.

Definition at line 299 of file telemetry_manager.h.

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

- lib/telemetry/telemetry_manager.h
- lib/telemetry/telemetry_manager.cpp

7.22 TelemetryRecord Struct Reference

Structure representing a single telemetry data point.

```
#include <telemetry_manager.h>
```

Public Member Functions

- std::string to_csv () const
  Converts the telemetry record to a CSV string.

Public Attributes

- uint32_t timestamp
- std::string build_version
- float battery_voltage
- float system_voltage
- float charge_current_usb
- float charge_current_solar
- float discharge_current
- std::string time
- std::string latitude
- std::string lat_dir
- std::string longitude
- std::string lon_dir
- std::string speed
- std::string course
- std::string date
- std::string fix_quality
- std::string satellites
- std::string altitude

7.22.1 Detailed Description

Structure representing a single telemetry data point.

Contains all measurements from power subsystem, sensors, and GPS data collected at a specific point in time

Definition at line 43 of file telemetry_manager.h.

7.22.2 Member Data Documentation

7.22.2.1 timestamp

`uint32_t TelemetryRecord::timestamp`

Unix timestamp of the record

Definition at line 44 of file telemetry_manager.h.

7.22.2.2 build_version

`std::string TelemetryRecord::build_version`

Build version of the firmware

Definition at line 46 of file telemetry_manager.h.

7.22.2.3 battery_voltage

`float TelemetryRecord::battery_voltage`

Battery voltage in volts

Definition at line 49 of file telemetry_manager.h.

7.22.2.4 system_voltage

`float TelemetryRecord::system_voltage`

System 5V rail voltage in volts

Definition at line 50 of file telemetry_manager.h.

7.22.2.5 charge_current_usb

`float TelemetryRecord::charge_current_usb`

USB charging current in mA

Definition at line 51 of file telemetry_manager.h.

7.22.2.6 charge_current_solar

```
float TelemetryRecord::charge_current_solar
```

Solar charging current in mA

Definition at line 52 of file telemetry_manager.h.

7.22.2.7 discharge_current

```
float TelemetryRecord::discharge_current
```

Battery discharge current in mA

Definition at line 53 of file telemetry_manager.h.

7.22.2.8 time

```
std::string TelemetryRecord::time
```

UTC time from GPS

Definition at line 56 of file telemetry_manager.h.

7.22.2.9 latitude

```
std::string TelemetryRecord::latitude
```

Latitude from GPS

Definition at line 57 of file telemetry_manager.h.

7.22.2.10 lat_dir

```
std::string TelemetryRecord::lat_dir
```

N/S latitude direction

Definition at line 58 of file telemetry_manager.h.

7.22.2.11 longitude

```
std::string TelemetryRecord::longitude
```

Longitude from GPS

Definition at line 59 of file telemetry_manager.h.

7.22.2.12 lon_dir

```
std::string TelemetryRecord::lon_dir
```

E/W longitude direction

Definition at line 60 of file telemetry_manager.h.

7.22.2.13 speed

```
std::string TelemetryRecord::speed
```

Speed in knots

Definition at line 61 of file telemetry_manager.h.

7.22.2.14 course

```
std::string TelemetryRecord::course
```

Course in degrees

Definition at line 62 of file telemetry_manager.h.

7.22.2.15 date

```
std::string TelemetryRecord::date
```

Date from GPS

Definition at line 63 of file telemetry_manager.h.

7.22.2.16 fix_quality

```
std::string TelemetryRecord::fix_quality
```

GPS fix quality

Definition at line 66 of file telemetry_manager.h.

7.22.2.17 satellites

```
std::string TelemetryRecord::satellites
```

Number of satellites in view

Definition at line 67 of file telemetry_manager.h.

7.22.2.18 altitude

```
std::string TelemetryRecord::altitude
```

Altitude in meters

Definition at line 68 of file telemetry_manager.h.

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

- lib/telemetry/telemetry_manager.h

Chapter 8

File Documentation

8.1 build_number.h File Reference

Macros

- #define BUILD_NUMBER 496

8.1.1 Macro Definition Documentation

8.1.1.1 BUILD_NUMBER

```
#define BUILD_NUMBER 496
```

Definition at line 6 of file build_number.h.

8.2 build_number.h

Go to the documentation of this file.

```
00001 //This file is automatically generated by build_number.cmake
00002
00003 #ifndef CMAKE_BUILD_NUMBER_HEADER
00004 #define CMAKE_BUILD_NUMBER_HEADER
00005
00006 #define BUILD_NUMBER 496
00007
00008 #endif
```

8.3 includes.h File Reference

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "hardware/i2c.h"
#include "hardware/uart.h"
#include "pico/multicore.h"
#include "event_manager.h"
#include "lib/powerman/PowerManager.h"
#include <pico/bootrom.h>
#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/clock/DS3231.h"
#include <iostream>
#include <iomanip>
#include <queue>
#include <chrono>
#include "protocol.h"
#include <atomic>
#include <map>
#include "pin_config.h"
#include "utils.h"
#include "communication.h"
#include "build_number.h"
#include "lib/location/gps_collector.h"
#include "lib/storage/storage.h"
#include "lib/storage/pico-vfs/include/filesystem/vfs.h"
#include "telemetry_manager.h"
#include "system_state_manager.h"
```

8.4 includes.h

Go to the documentation of this file.

```
#ifndef INCLUDES_H
#define INCLUDES_H

#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "hardware/i2c.h"
#include "hardware/uart.h"
#include "pico/multicore.h"
#include "event_manager.h"
#include "lib/powerman/PowerManager.h"
#include <pico/bootrom.h>

#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/clock/DS3231.h"
#include <iostream>
#include <iomanip>
#include <queue>
#include <chrono>
#include "protocol.h"
#include <atomic>
#include <iostream>
#include <map>
#include "pin_config.h"
#include "utils.h"
#include "communication.h"
```

```
#include "build_number.h"
#include "lib/location/gps_collector.h"
#include "lib/storage/storage.h"
#include "lib/storage/pico-vfs/include/filesystem/vfs.h"
#include "telemetry_manager.h"
#include "system_state_manager.h"

#endif
```

8.5 lib/clock/DS3231.cpp File Reference

```
#include "DS3231.h"
#include "utils.h"
#include <cstdio>
#include <mutex>
#include "event_manager.h"
#include "NMEA_data.h"
```

8.6 DS3231.cpp

Go to the documentation of this file.

```
#include "DS3231.h"
#include "utils.h"
#include <cstdio>
#include <mutex>
#include "event_manager.h"
#include "NMEA_data.h"


DS3231::DS3231() : i2c(MAIN_I2C_PORT), ds3231_addr(DS3231_DEVICE_ADRESS) {
    recursive_mutex_init(&clock_mutex_);
}


DS3231& DS3231::get_instance() {
    static DS3231 instance;
    return instance;
}

int DS3231::set_time(ds3231_data_t *data) {
    uint8_t temp[7] = {0};

    if (clock_enable() != 0) {
        uart_print("Failed to enable clock oscillator", VerbosityLevel::ERROR);
        return -1;
    }

    if (data->seconds > 59)
        data->seconds = 59;
    if (data->minutes > 59)
        data->minutes = 59;
    if (data->hours > 23)
        data->hours = 23;
    if (data->day > 7)
        data->day = 7;
    else if (data->day < 1)
        data->day = 1;
    if (data->date > 31)
        data->date = 31;
    else if (data->date < 1)
        data->date = 1;
    if (data->month > 12)
        data->month = 12;
    else if (data->month < 1)
        data->month = 1;
    if (data->year > 99)
        data->year = 99;

    temp[0] = bin_to_bcd(data->seconds);
    temp[1] = bin_to_bcd(data->minutes);
    temp[2] = bin_to_bcd(data->hours);
```

```
    temp[2] &= ~(0x01 « 6); // Clear 12/24 hour bit
    temp[3] = bin_to_bcd(data->day);
    temp[4] = bin_to_bcd(data->date);
    temp[5] = bin_to_bcd(data->month);
    if (data->century)
        temp[5] |= (0x01 « 7);
    temp[6] = bin_to_bcd(data->year);

    std::string status = "BCD values to be written to DS3231: " + std::to_string(temp[0]) + " " +
                        std::to_string(temp[1]) + " " + std::to_string(temp[2]) + " " +
                        std::to_string(temp[3]) + " " + std::to_string(temp[4]) + " " +
                        std::to_string(temp[5]) + " " + std::to_string(temp[6]);

    uart_print(status, VerbosityLevel::DEBUG);

    int result = i2c_write_reg(DS3231_SECONDS_REG, 7, temp);
    if (result != 0) {
        uart_print("i2c write failed", VerbosityLevel::ERROR);
        return -1;
    }

    return 0;
}


int DS3231::get_time(ds3231_data_t *data) {
    std::string status;
    uint8_t raw_data[7];
    int result = i2c_read_reg(DS3231_SECONDS_REG, 7, raw_data);
    if (result != 0) {
        status = "Failed to read time from DS3231";
        uart_print(status, VerbosityLevel::ERROR);
        return -1;
    }

    data->seconds = bcd_to_bin(raw_data[0] & 0x7F); // Masking for CH bit (clock halt)
    data->minutes = bcd_to_bin(raw_data[1] & 0x7F);
    data->hours = bcd_to_bin(raw_data[2] & 0x3F);   // Masking for 12/24 hour mode bit
    data->day = raw_data[3] & 0x07;                  // Day of week (1-7)
    data->date = bcd_to_bin(raw_data[4] & 0x3F);
    data->month = bcd_to_bin(raw_data[5] & 0x1F);   // Masking for century bit
    data->century = (raw_data[5] & 0x80) » 7;
    data->year = bcd_to_bin(raw_data[6]);

    if (data->seconds > 59 || data->minutes > 59 || data->hours > 23 ||
        data->day < 1 || data->day > 7 || data->date < 1 || data->date > 31 ||
        data->month < 1 || data->month > 12 || data->year > 99) {
        uart_print("Invalid data read from DS3231", VerbosityLevel::ERROR);
        return -1;
    }

    return 0;
}


int DS3231::read_temperature(float *resolution) {
    std::string status;
    uint8_t temp[2];
    int result = i2c_read_reg(DS3231_TEMPERATURE_MSB_REG, 2, temp);
    if (result != 0) {
        status = "Failed to read temperature from DS3231";
        uart_print(status, VerbosityLevel::ERROR);
        return -1;
    }

    int8_t temperature_msb = (int8_t)temp[0];
    uint8_t temperature_lsb = temp[1] » 6;    // Only the 2 MSB are valid

    *resolution = temperature_msb + (temperature_lsb * 0.25f); // 0.25 degree resolution

    return 0;
}


int DS3231::set_unix_time(time_t unix_time) {
    struct tm *timeinfo = gmtime(&unix_time);
    if (timeinfo == NULL) {
        uart_print("Error: gmtime() failed", VerbosityLevel::ERROR);
        return -1;
    }

    ds3231_data_t data;
    data.seconds = timeinfo->tm_sec;
    data.minutes = timeinfo->tm_min;
    data.hours = timeinfo->tm_hour;
    data.day = timeinfo->tm_wday == 0 ? 7 : timeinfo->tm_wday; // Sunday is 0 in tm struct, but 1 in DS3231
```

```
    data.date = timeinfo->tm_mday;
    data.month = timeinfo->tm_mon + 1; // Month is 0-11 in tm struct, but 1-12 in DS3231
    data.year = timeinfo->tm_year - 100; // Year is since 1900, we want the last two digits
    data.century = timeinfo->tm_year >= 2000;

    return set_time(&data);
}


time_t DS3231::get_unix_time() {
    ds3231_data_t data;
    if (get_time(&data)) {
        return -1;
    }

    struct tm timeinfo;
    timeinfo.tm_sec = data.seconds;
    timeinfo.tm_min = data.minutes;
    timeinfo.tm_hour = data.hours;
    timeinfo.tm_mday = data.date;
    timeinfo.tm_mon = data.month - 1; // Month is 0-11 in tm struct, but 1-12 in DS3231
    timeinfo.tm_year = data.year + 100; // Year is since 1900

    // mktime assumes that tm_wday and tm_yday are uninitialized
    timeinfo.tm_wday = 0;
    timeinfo.tm_yday = 0;
    timeinfo.tm_isdst = 0; // Set to 0 to use UTC

    time_t timestamp = mktime(&timeinfo);
    if (timestamp == (time_t)(-1)) {
        uart_print("Error: mktime() failed", VerbosityLevel::ERROR);
        return -1;
    }

    return timestamp;
}


int DS3231::clock_enable() {
    std::string status;
    uint8_t control_reg = 0;
    int result = i2c_read_reg(DS3231_CONTROL_REG, 1, &control_reg);
    if (result != 0) {
        status = "Failed to read control register";
        uart_print(status, VerbosityLevel::ERROR);
        return -1;
    }

    // Clear the EOSC bit to enable the oscillator
    control_reg &= ~(1 « 7);

    result = i2c_write_reg(DS3231_CONTROL_REG, 1, &control_reg);
    if (result != 0) {
        status = "Failed to write control register";
        uart_print(status, VerbosityLevel::ERROR);
        return -1;
    }

    return 0;
}


int16_t DS3231::get_timezone_offset() const {
    return timezone_offset_minutes_;
}


void DS3231::set_timezone_offset(int16_t offset_minutes) {
    // Validate range: -12 hours to +12 hours (-720 to +720 minutes)
    if (offset_minutes >= -720 && offset_minutes <= 720) {
        timezone_offset_minutes_ = offset_minutes;
    } else {
        uart_print("Error: Invalid timezone offset", VerbosityLevel::ERROR);
    }
}


uint32_t DS3231::get_clock_sync_interval() const {
    return sync_interval_minutes_;
}


void DS3231::set_clock_sync_interval(uint32_t interval_minutes) {
    if (interval_minutes >= 1 && interval_minutes <= 43200) {
        sync_interval_minutes_ = interval_minutes;
    } else {
        uart_print("Error: Invalid sync interval", VerbosityLevel::ERROR);
```

```
    }
}


time_t DS3231::get_last_sync_time() const {
    return last_sync_time_;
}


void DS3231::update_last_sync_time() {
    last_sync_time_ = get_unix_time();
    uart_print("Clock sync time updated: " + std::to_string(last_sync_time_), VerbosityLevel::INFO);
}


time_t DS3231::get_local_time() {
    time_t utc_time = get_unix_time();
    if (utc_time == -1) {
        return -1;
    }

    return utc_time + (timezone_offset_minutes_ * 60);
}


bool DS3231::is_sync_needed() {
    if (last_sync_time_ == 0) {
        return true;
    }

    time_t current_time = get_unix_time();
    if (current_time == -1) {
        return true;
    }

    time_t time_since_last_sync = current_time - last_sync_time_;
    uint32_t minutes_since_last_sync = time_since_last_sync / 60;

    return minutes_since_last_sync >= sync_interval_minutes_;
}


bool DS3231::sync_clock_with_gps() {
    auto& nmea_data = NMEAData::get_instance();

    if (!nmea_data.has_valid_time()) {
        uart_print("GPS time data not available for sync", VerbosityLevel::WARNING);
        EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
        return false;
    }

    time_t gps_time = nmea_data.get_unix_time();
    if (gps_time <= 0) {
        uart_print("Invalid GPS time for sync", VerbosityLevel::ERROR);
        EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
        return false;
    }

    if (set_unix_time(gps_time) != 0) {
        uart_print("Failed to set system time from GPS", VerbosityLevel::ERROR);
        EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
        return false;
    }

    update_last_sync_time();

    EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC);
    uart_print("Clock synced with GPS time: " + std::to_string(gps_time), VerbosityLevel::INFO);

    return true;
}

// ==================== private methods

int DS3231::i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
    if (!length)
        return -1;

    std::string status = "Reading register " + std::to_string(reg_addr) + " from DS3231";
    uart_print(status, VerbosityLevel::DEBUG);
    recursive_mutex_enter_blocking(&clock_mutex_);
    uint8_t reg = reg_addr;
    int write_result = i2c_write_blocking(i2c, ds3231_addr, &reg, 1, true);
    if (write_result == PICO_ERROR_GENERIC) {
        status = "Failed to write register address to DS3231";
        uart_print(status, VerbosityLevel::ERROR);
        recursive_mutex_exit(&clock_mutex_);
```

```
        return -1;
    }
    int read_result = i2c_read_blocking(i2c, ds3231_addr, data, length, false);
    if (read_result == PICO_ERROR_GENERIC) {
        status = "Failed to read register data from DS3231";
        uart_print(status, VerbosityLevel::ERROR);
        recursive_mutex_exit(&clock_mutex_);
        return -1;
    }
    recursive_mutex_exit(&clock_mutex_);

    return 0;
}

int DS3231::i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
    if (!length)
        return -1;

    recursive_mutex_enter_blocking(&clock_mutex_);
    std::vector<uint8_t> message(length + 1);
    message[0] = reg_addr;
    for (size_t i = 0; i < length; i++) {
        message[i + 1] = data[i];
    }
    int write_result = i2c_write_blocking(i2c, ds3231_addr, message.data(), (length + 1), false);
    if (write_result == PICO_ERROR_GENERIC) {
        uart_print("Error: i2c_write_blocking failed in i2c_write_reg", VerbosityLevel::ERROR);
        recursive_mutex_exit(&clock_mutex_);
        return -1;
    }
    recursive_mutex_exit(&clock_mutex_);

    return 0;
}

uint8_t DS3231::bin_to_bcd(const uint8_t data) {
    uint8_t ones_digit = (uint8_t)(data % 10);
    uint8_t tens_digit = (uint8_t)(data - ones_digit) / 10;
    return ((tens_digit « 4) + ones_digit);
}


uint8_t DS3231::bcd_to_bin(const uint8_t bcd) {
    uint8_t ones_digit = (uint8_t)(bcd & 0x0F);
    uint8_t tens_digit = (uint8_t)(bcd » 4);
    return (tens_digit * 10 + ones_digit);
} // End of DS3231_RTC group
```

8.7 lib/clock/DS3231.h File Reference

```
#include <string>
#include <array>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include <time.h>
#include "pico/mutex.h"
#include "lib/location/gps_collector.h"
```

Classes

- struct ds3231_data_t

  Structure to hold time and date information from DS3231.
- class DS3231

  Class for interfacing with the DS3231 real-time clock.

Macros

- #define DS3231_DEVICE_ADRESS 0x68
    DS3231 I2C device address.
- #define DS3231_SECONDS_REG 0x00
    Register address: Seconds (0-59)
- #define DS3231_MINUTES_REG 0x01
    Register address: Minutes (0-59)
- #define DS3231_HOURS_REG 0x02
    Register address: Hours (0-23 in 24hr mode)
- #define DS3231_DAY_REG 0x03
    Register address: Day of the week (1-7)
- #define DS3231_DATE_REG 0x04
    Register address: Date (1-31)
- #define DS3231_MONTH_REG 0x05
    Register address: Month (1-12) & Century bit.
- #define DS3231_YEAR_REG 0x06
    Register address: Year (00-99)
- #define DS3231_CONTROL_REG 0x0E
    Register address: Control register.
- #define DS3231_CONTROL_STATUS_REG 0x0F
    Register address: Control/Status register.
- #define DS3231_TEMPERATURE_MSB_REG 0x11
    Register address: Temperature register (MSB)
- #define DS3231_TEMPERATURE_LSB_REG 0x12
    Register address: Temperature register (LSB)

Enumerations

- enum days_of_week {
  MONDAY = 1 , TUESDAY , WEDNESDAY , THURSDAY ,
  FRIDAY , SATURDAY , SUNDAY }
    Enumeration of days of the week.

8.7.1 Macro Definition Documentation

8.7.1.1 DS3231_DEVICE_ADRESS

```
#define DS3231_DEVICE_ADRESS 0x68
```

DS3231 I2C device address.

Definition at line 15 of file DS3231.h.

8.7.1.2 DS3231_SECONDS_REG

```
#define DS3231_SECONDS_REG 0x00
```

Register address: Seconds (0-59)

Definition at line 20 of file DS3231.h.

8.7.1.3 DS3231_MINUTES_REG

```
#define DS3231_MINUTES_REG 0x01
```

Register address: Minutes (0-59)

Definition at line 25 of file DS3231.h.

8.7.1.4 DS3231_HOURS_REG

```
#define DS3231_HOURS_REG 0x02
```

Register address: Hours (0-23 in 24hr mode)

Definition at line 30 of file DS3231.h.

8.7.1.5 DS3231_DAY_REG

```
#define DS3231_DAY_REG 0x03
```

Register address: Day of the week (1-7)

Definition at line 35 of file DS3231.h.

8.7.1.6 DS3231_DATE_REG

```
#define DS3231_DATE_REG 0x04
```

Register address: Date (1-31)

Definition at line 40 of file DS3231.h.

8.7.1.7 DS3231_MONTH_REG

```
#define DS3231_MONTH_REG 0x05
```

Register address: Month (1-12) & Century bit.

Definition at line 45 of file DS3231.h.

8.7.1.8 DS3231_YEAR_REG

```
#define DS3231_YEAR_REG 0x06
```

Register address: Year (00-99)

Definition at line 50 of file DS3231.h.

8.7.1.9 DS3231_CONTROL_REG

```
#define DS3231_CONTROL_REG 0x0E
```

Register address: Control register.

Definition at line 55 of file DS3231.h.

8.7.1.10 DS3231_CONTROL_STATUS_REG

```
#define DS3231_CONTROL_STATUS_REG 0x0F
```

Register address: Control/Status register.

Definition at line 60 of file DS3231.h.

8.7.1.11 DS3231_TEMPERATURE_MSB_REG

```
#define DS3231_TEMPERATURE_MSB_REG 0x11
```

Register address: Temperature register (MSB)

Definition at line 65 of file DS3231.h.

8.7.1.12 DS3231_TEMPERATURE_LSB_REG

```
#define DS3231_TEMPERATURE_LSB_REG 0x12
```

Register address: Temperature register (LSB)

Definition at line 70 of file DS3231.h.

8.7.2 Enumeration Type Documentation

8.7.2.1 days_of_week

```
enum days_of_week
```

Enumeration of days of the week.

Enumerator

MONDAY	Monday.
TUESDAY	Tuesday.
WEDNESDAY	Wednesday.
THURSDAY	Thursday.
FRIDAY	Friday.
SATURDAY	Saturday.
SUNDAY	Sunday.

Definition at line 76 of file DS3231.h.

8.8 DS3231.h

Go to the documentation of this file.

```
#ifndef DS3231_H
#define DS3231_H

#include <string>
#include <array>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include <time.h>
#include "pico/mutex.h"
#include "lib/location/gps_collector.h"

#define DS3231_DEVICE_ADRESS            0x68

#define DS3231_SECONDS_REG              0x00

#define DS3231_MINUTES_REG              0x01

#define DS3231_HOURS_REG                0x02

#define DS3231_DAY_REG                  0x03

#define DS3231_DATE_REG                 0x04

#define DS3231_MONTH_REG                0x05

#define DS3231_YEAR_REG                 0x06

#define DS3231_CONTROL_REG              0x0E

#define DS3231_CONTROL_STATUS_REG       0x0F

#define DS3231_TEMPERATURE_MSB_REG      0x11

#define DS3231_TEMPERATURE_LSB_REG      0x12

enum days_of_week {
    MONDAY  = 1,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

typedef struct {
    uint8_t seconds;
    uint8_t minutes;
    uint8_t hours;
    uint8_t day;
    uint8_t date;
    uint8_t month;
    uint8_t year;
    bool century;
} ds3231_data_t;

class DS3231 {
public:
    DS3231(i2c_inst_t *i2c_instance);
    static DS3231& get_instance();

    int set_time(ds3231_data_t *data);

    int get_time(ds3231_data_t *data);

    int read_temperature(float *resolution);

    int set_unix_time(time_t unix_time);

    time_t get_unix_time();

    int clock_enable();

    int16_t get_timezone_offset() const;

    void set_timezone_offset(int16_t offset_minutes);

    uint32_t get_clock_sync_interval() const;

    void set_clock_sync_interval(uint32_t interval_minutes);

    time_t get_last_sync_time() const;
```

```
    void update_last_sync_time();

    time_t get_local_time();

    bool is_sync_needed();

    bool sync_clock_with_gps();


private:
    i2c_inst_t *i2c;
    uint8_t ds3231_addr;
    recursive_mutex_t clock_mutex_;
    int16_t timezone_offset_minutes_ = 60;
    uint32_t sync_interval_minutes_ = 1440;
    time_t last_sync_time_ = 0;

    // Private constructor
    DS3231();

    // Delete copy constructor and assignment operator
    DS3231(const DS3231&) = delete;
    DS3231& operator=(const DS3231&) = delete;

    int i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data);

    int i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data);

    uint8_t bin_to_bcd(const uint8_t data);

    uint8_t bcd_to_bin(const uint8_t bcd);
};

#endif // DS3231_H
```

8.9 lib/comms/commands/clock_commands.cpp File Reference

```
#include "communication.h"
#include <time.h>
#include "DS3231.h"
```

Macros

- #define CLOCK_GROUP 3
- #define TIME 0
- #define TIMEZONE_OFFSET 1
- #define CLOCK_SYNC_INTERVAL 2
- #define LAST_SYNC_TIME 3

Functions

- std::vector< Frame > handle_time (const std::string &param, OperationType operationType)

    Handler for getting and setting system time.
- std::vector< Frame > handle_timezone_offset (const std::string &param, OperationType operationType)

    Handler for getting and setting timezone offset.
- std::vector< Frame > handle_clock_sync_interval (const std::string &param, OperationType operationType)

    Handler for getting and setting clock synchronization interval.
- std::vector< Frame > handle_get_last_sync_time (const std::string &param, OperationType operationType)

    Handler for getting last clock sync time.

8.9.1 Macro Definition Documentation

8.9.1.1 CLOCK_GROUP

```
#define CLOCK_GROUP 3
```

Definition at line 5 of file clock_commands.cpp.

8.9.1.2 TIME

```
#define TIME 0
```

Definition at line 6 of file clock_commands.cpp.

8.9.1.3 TIMEZONE_OFFSET

```
#define TIMEZONE_OFFSET 1
```

Definition at line 7 of file clock_commands.cpp.

8.9.1.4 CLOCK_SYNC_INTERVAL

```
#define CLOCK_SYNC_INTERVAL 2
```

Definition at line 8 of file clock_commands.cpp.

8.9.1.5 LAST_SYNC_TIME

```
#define LAST_SYNC_TIME 3
```

Definition at line 9 of file clock_commands.cpp.

8.10 clock_commands.cpp

Go to the documentation of this file.

```
#include "communication.h"
#include <time.h>
#include "DS3231.h" // Include the DS3231 header

#define CLOCK_GROUP 3
#define TIME 0
#define TIMEZONE_OFFSET 1
#define CLOCK_SYNC_INTERVAL 2
#define LAST_SYNC_TIME 3



std::vector<Frame> handle_time(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType == OperationType::SET) {
        if (param.empty()) {
            error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
            return frames;
        }
        try {
            time_t newTime = std::stoll(param);
            if (newTime <= 0) {
                error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
                frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
                return frames;
            }

            if (DS3231::get_instance().set_unix_time(newTime) != 0) {
                error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
                frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
                return frames;
            }

            EventEmitter::emit(EventGroup::CLOCK, ClockEvent::CHANGED);
            frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, TIME, std::to_string(DS3231::get_instance().get_unix_time())));
            return frames;
        } catch (...) {
            error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
            return frames;
        }
    } else if (operationType == OperationType::GET) {
        if (!param.empty()) {
            error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
            return frames;
        }

        uint32_t time_unix = DS3231::get_instance().get_local_time();
        if (time_unix == 0) {
            error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
            return frames;
        }

        frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, TIME, std::to_string(time_unix)));
        return frames;
    }

    error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
    frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
    return frames;
}

std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!(operationType == OperationType::GET || operationType == OperationType::SET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
        return frames;
    }

    if (operationType == OperationType::GET) {
        if (!param.empty()) {
            error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
```

```
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
            return frames;
        }

        int offset = DS3231::get_instance().get_timezone_offset();
        std::string offset_set = std::to_string(offset);
        frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, TIMEZONE_OFFSET, offset_set));
        return frames;
    }

    if (param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
        frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
        return frames;
    }

    try {
        int16_t offset = std::stoi(param);
        if (offset < -720 || offset > 720) {
            error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
            return frames;
        }

        DS3231::get_instance().set_timezone_offset(offset);
        std::string offset_set = std::to_string(offset);
        frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, TIMEZONE_OFFSET, offset_set));
        return frames;
    } catch (...) {
        error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
        frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
        return frames;
    }
}


std::vector<Frame> handle_clock_sync_interval(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!(operationType == OperationType::GET || operationType == OperationType::SET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
        return frames;
    }

    if (operationType == OperationType::GET) {
        if (!param.empty()) {
            error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
            return frames;
        }

        uint32_t syncInterval = DS3231::get_instance().get_clock_sync_interval();
        std::string clockSyncInterval = std::to_string(syncInterval);
        frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, clockSyncInterval));
        return frames;
    }

    if (operationType == OperationType::SET) {
        if (param.empty()) {
            error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
            return frames;
        }
        try {
            uint32_t interval = std::stoul(param);

            DS3231::get_instance().set_clock_sync_interval(interval);
            std::string interval_set = std::to_string(interval);

            frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, interval_set));
            return frames;
        } catch (...) {
            error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
            frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
        }
    }
    error_msg = error_code_to_string(ErrorCode::UNKNOWN_ERROR);
```

```
    frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
    return frames;
}

std::vector<Frame> handle_get_last_sync_time(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType != OperationType::GET || !param.empty()) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, LAST_SYNC_TIME, error_msg));
        return frames;
    }

    time_t lastSyncTime = DS3231::get_instance().get_last_sync_time();

    if (lastSyncTime == 0) {
        frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, LAST_SYNC_TIME, "NEVER"));
    } else {
        frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, LAST_SYNC_TIME,
                         std::to_string(lastSyncTime)));
    }

    return frames;
} // end of ClockCommands group
```

8.11 lib/comms/commands/commands.cpp File Reference

```
#include "commands.h"
#include "communication.h"
```

Typedefs

- using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>

  Function type for command handlers.
- using CommandMap = std::map<uint32_t, CommandHandler>

  Map type for storing command handlers.

Functions

- std::vector< Frame > execute_command (uint32_t commandKey, const std::string &param, OperationType operationType)

  Executes a command based on its key.

Variables

- CommandMap command_handlers

  Global map of all command handlers.

8.12 commands.cpp

Go to the documentation of this file.

```
// commands/commands.cpp
#include "commands.h"
#include "communication.h"


using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>;

using CommandMap = std::map<uint32_t, CommandHandler>;

CommandMap command_handlers = {
    {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(0)), handle_get_commands_list}, // Group 1, Command 0
    {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(1)), handle_get_build_version}, // Group 1, Command 1
    {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(8)), handle_verbosity}, // Group 1, Command 9
    {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(9)), handle_enter_bootloader_mode}, // Group 1, Command 9
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(0)), handle_get_power_manager_ids}, // Group 2, Command 0
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(2)), handle_get_voltage_battery}, // Group 2, Command 2
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(3)), handle_get_voltage_5v}, // Group 2, Command 3
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(4)), handle_get_current_charge_usb}, // Group 2, Command 4
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(5)), handle_get_current_charge_solar}, // Group 2, Command 5
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(6)), handle_get_current_charge_total}, // Group 2, Command 6
    {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(7)), handle_get_current_draw}, // Group 2, Command 7
    {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(0)), handle_time}, // Group 3, Command 0
    {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(1)), handle_timezone_offset}, // Group 3, Command 1
    {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(2)), handle_clock_sync_interval}, // Group 3, Command 2
    {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(3)), handle_get_last_sync_time}, // Group 3, Command 3
    {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(0)), handle_get_sensor_data}, // Group 4, Command 0
    {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(1)), handle_sensor_config}, // Group 4, Command 1
    {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(3)), handle_get_sensor_list}, // Group 4, Command 3
    {((static_cast<uint32_t>(5) << 8) | static_cast<uint32_t>(1)), handle_get_last_events}, // Group 5, Command 1
    {((static_cast<uint32_t>(5) << 8) | static_cast<uint32_t>(2)), handle_get_event_count}, // Group 5, Command 2
    {((static_cast<uint32_t>(6) << 8) | static_cast<uint32_t>(0)), handle_list_files}, // Group 6, Command 0
    {((static_cast<uint32_t>(6) << 8) | static_cast<uint32_t>(4)), handle_mount}, // Group 6, Command 4
    {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(1)), handle_gps_power_status}, // Group 7, Command 1
    {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(2)), handle_enable_gps_uart_passthrough}, // Group 7, Command 3
    {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(3)), handle_get_rmc_data}, // Group 7, Command 3
    {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(4)), handle_get_gga_data}, // Group 7, Command 4
    {((static_cast<uint32_t>(8) << 8) | static_cast<uint32_t>(2)), handle_get_last_telemetry_record},
    {((static_cast<uint32_t>(8) << 8) | static_cast<uint32_t>(3)), handle_get_last_sensor_record},
};


std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
operationType) {
    auto it = command_handlers.find(commandKey);
    if (it != command_handlers.end()) {
        CommandHandler handler = it->second;
        return handler(param, operationType);
    } else {
        std::vector<Frame> frames;
        frames.push_back(frame_build(OperationType::ERR, 0, 0, "INVALID COMMAND"));
        return frames;
    }
} // end of CommandSystem group
```

8.13 lib/comms/commands/commands.h File Reference

```
#include <string>
#include <functional>
#include <map>
#include "protocol.h"
```

Functions

- std::vector< Frame > handle_time (const std::string &param, OperationType operationType)

  Handler for getting and setting system time.
- std::vector< Frame > handle_timezone_offset (const std::string &param, OperationType operationType)

  Handler for getting and setting timezone offset.
- std::vector< Frame > handle_clock_sync_interval (const std::string &param, OperationType operationType)

  Handler for getting and setting clock synchronization interval.
- std::vector< Frame > handle_get_last_sync_time (const std::string &param, OperationType operationType)

  Handler for getting last clock sync time.
- std::vector< Frame > handle_get_commands_list (const std::string &param, OperationType operationType)

  Handler for listing all available commands on UART.
- std::vector< Frame > handle_get_build_version (const std::string &param, OperationType operationType)

  Get firmware build version.
- std::vector< Frame > handle_verbosity (const std::string &param, OperationType operationType)

  Handles setting or getting the UART verbosity level.
- std::vector< Frame > handle_enter_bootloader_mode (const std::string &param, OperationType operationType)

  Reboot system to USB firmware loader.
- std::vector< Frame > handle_gps_power_status (const std::string &param, OperationType operationType)

  Handler for controlling GPS module power state.
- std::vector< Frame > handle_enable_gps_uart_passthrough (const std::string &param, OperationType operationType)

  Handler for enabling GPS transparent mode (UART pass-through)
- std::vector< Frame > handle_get_rmc_data (const std::string &param, OperationType operationType)

  Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.
- std::vector< Frame > handle_get_gga_data (const std::string &param, OperationType operationType)

  Handler for retrieving GPS GGA (Global Positioning System Fix Data) data.
- std::vector< Frame > handle_get_power_manager_ids (const std::string &param, OperationType operationType)

  Handler for retrieving Power Manager IDs.
- std::vector< Frame > handle_get_voltage_battery (const std::string &param, OperationType operationType)

  Handler for getting battery voltage.
- std::vector< Frame > handle_get_voltage_5v (const std::string &param, OperationType operationType)

  Handler for getting 5V rail voltage.
- std::vector< Frame > handle_get_current_charge_usb (const std::string &param, OperationType operationType)

  Handler for getting USB charge current.
- std::vector< Frame > handle_get_current_charge_solar (const std::string &param, OperationType operationType)

  Handler for getting solar panel charge current.
- std::vector< Frame > handle_get_current_charge_total (const std::string &param, OperationType operationType)

Handler for getting total charge current.

- std::vector< Frame > handle_get_current_draw (const std::string &param, OperationType operationType)

  Handler for getting system current draw.
- std::vector< Frame > handle_get_last_events (const std::string &param, OperationType operationType)

  Handler for retrieving last N events from the event log.
- std::vector< Frame > handle_get_event_count (const std::string &param, OperationType operationType)

  Handler for getting total number of events in the log.
- std::vector< Frame > handle_list_files (const std::string &param, OperationType operationType)

  Handles the list files command.
- std::vector< Frame > handle_mount (const std::string &param, OperationType operationType)

  Handles the SD card mount/unmount command.
- std::vector< Frame > handle_get_sensor_data (const std::string &param, OperationType operationType)

  Handler for reading sensor data.
- std::vector< Frame > handle_sensor_config (const std::string &param, OperationType operationType)

  Handler for configuring sensors.
- std::vector< Frame > handle_get_sensor_list (const std::string &param, OperationType operationType)

  Handler for listing available sensors.
- std::vector< Frame > handle_get_last_telemetry_record (const std::string &param, OperationType operationType)

  Handles the get last record command.
- std::vector< Frame > handle_get_last_sensor_record (const std::string &param, OperationType operationType)

  Handles the get last sensor record command.
- std::vector< Frame > execute_command (uint32_t commandKey, const std::string &param, OperationType operationType)

  Executes a command based on its key.

Variables

- std::map< uint32_t, std::function< std::vector< Frame >(const std::string &, OperationType)> > command_handlers

  Global map of all command handlers.

8.14 commands.h

Go to the documentation of this file.

```
// commands/commands.h
#ifndef COMMANDS_H
#define COMMANDS_H

#include <string>
#include <functional>
#include <map>
#include "protocol.h"

// CLOCK
std::vector<Frame> handle_time(const std::string& param, OperationType operationType);
std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType);
std::vector<Frame> handle_clock_sync_interval(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_last_sync_time(const std::string& param, OperationType operationType);


// DIAG
std::vector<Frame> handle_get_commands_list(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_build_version(const std::string& param, OperationType operationType);
std::vector<Frame> handle_verbosity(const std::string& param, OperationType operationType);
std::vector<Frame> handle_enter_bootloader_mode(const std::string& param, OperationType
    operationType);
```

```


// GPS
std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType);
std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_rmc_data(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_gga_data(const std::string& param, OperationType operationType);


// POWER
std::vector<Frame> handle_get_power_manager_ids(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_voltage_battery(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_voltage_5v(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_current_charge_usb(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_current_charge_solar(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_current_charge_total(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_current_draw(const std::string& param, OperationType operationType);


// EVENT
std::vector<Frame> handle_get_last_events(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType);


//STORAGE
std::vector<Frame> handle_list_files(const std::string& param, OperationType operationType);
std::vector<Frame> handle_mount(const std::string& param, OperationType operationType);

// SENSOR
std::vector<Frame> handle_get_sensor_data(const std::string& param, OperationType operationType);
std::vector<Frame> handle_sensor_config(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_sensor_list(const std::string& param, OperationType operationType);


// TELEMETRY
std::vector<Frame> handle_get_last_telemetry_record(const std::string& param, OperationType operationType);
std::vector<Frame> handle_get_last_sensor_record(const std::string& param, OperationType operationType);

std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType operationType);
extern std::map<uint32_t, std::function<std::vector<Frame>(const std::string&, OperationType)»
command_handlers;

#endif
```

8.15 lib/comms/commands/diagnostic_commands.cpp File Reference

```
#include "communication.h"
#include "commands.h"
#include "pico/stdlib.h"
#include "pico/bootrom.h"
#include "system_state_manager.h"
```

Functions

- std::vector< Frame > handle_get_commands_list (const std::string &param, OperationType operationType)

    Handler for listing all available commands on UART.
- std::vector< Frame > handle_get_build_version (const std::string &param, OperationType operationType)

    Get firmware build version.
- std::vector< Frame > handle_verbosity (const std::string &param, OperationType operationType)

    Handles setting or getting the UART verbosity level.
- std::vector< Frame > handle_enter_bootloader_mode (const std::string &param, OperationType operationType)

    Reboot system to USB firmware loader.

8.16 diagnostic_commands.cpp

Go to the documentation of this file.

```
#include "communication.h"
#include "commands.h"
#include "pico/stdlib.h"
#include "pico/bootrom.h"
#include "system_state_manager.h"

std::vector<Frame> handle_get_commands_list(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, 1, 0, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, 1, 0, error_msg));
        return frames;
    }

    std::string combined_command_details;
    for (const auto& entry : command_handlers) {
        uint32_t command_key = entry.first;
        uint8_t group = (command_key » 8) & 0xFF;
        uint8_t command = command_key & 0xFF;

        std::string command_details = std::to_string(group) + "." + std::to_string(command);

        if (combined_command_details.length() + command_details.length() + 1 > 100) {
            frames.push_back(frame_build(OperationType::SEQ, 1, 0, combined_command_details));
            combined_command_details = "";
        }

        if (!combined_command_details.empty()) {
            combined_command_details += "-";
        }
        combined_command_details += command_details;
    }

    if (!combined_command_details.empty()) {
        frames.push_back(frame_build(OperationType::SEQ, 1, 0, combined_command_details));
    }

    frames.push_back(frame_build(OperationType::VAL, 1, 0, "SEQ_DONE"));
    return frames;
}


std::vector<Frame> handle_get_build_version(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, 1, 1, error_msg));
        return frames;
    }

    if (operationType == OperationType::GET) {
        frames.push_back(frame_build(OperationType::VAL, 1, 1, std::to_string(BUILD_NUMBER)));
        return frames;
    }

    error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
    frames.push_back(frame_build(OperationType::ERR, 1, 1, error_msg));
    return frames;
}


std::vector<Frame> handle_verbosity(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType == OperationType::GET && param.empty()) {
        VerbosityLevel current_level = SystemStateManager::get_instance().get_uart_verbosity();
        uart_print("GET_VERBOSITY_ " + std::to_string(static_cast<int>(current_level)),
                   VerbosityLevel::INFO);
        frames.push_back(frame_build(OperationType::VAL, 1, 8,
                                     std::to_string(static_cast<int>(current_level))));
        return frames;
```

```
    }

    try {
        int level = std::stoi(param);
        if (level < 0 || level > 5) {
            error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
            frames.push_back(frame_build(OperationType::ERR, 1, 8, error_msg));
            return frames;
        }
        SystemStateManager::get_instance().set_uart_verbosity(static_cast<VerbosityLevel>(level));
        frames.push_back(frame_build(OperationType::RES, 1, 8, "LEVEL SET"));
        return frames;
    } catch (...) {
        error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
        frames.push_back(frame_build(OperationType::ERR, 1, 8, error_msg));
        return frames;
    }
}

std::vector<Frame> handle_enter_bootloader_mode(const std::string& param, OperationType operationType)
{
    std::vector<Frame> frames;
    std::string error_msg;

    if (param != "USB") {
        error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
        frames.push_back(frame_build(OperationType::ERR, 1, 9, error_msg));
        return frames;
    }

    if (operationType != OperationType::SET) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, 1, 9, error_msg));
        return frames;
    }

    frames.push_back(frame_build(OperationType::RES, 1, 9, "REBOOT BOOTSEL"));

    SystemStateManager::get_instance().set_bootloader_reset_pending(true);

    return frames;
}
```

8.17 lib/comms/commands/event_commands.cpp File Reference

```
#include "communication.h"
#include "event_manager.h"
#include <sstream>
```

Functions

- std::vector< Frame > handle_get_last_events (const std::string &param, OperationType operationType)

  Handler for retrieving last N events from the event log.
- std::vector< Frame > handle_get_event_count (const std::string &param, OperationType operationType)

  Handler for getting total number of events in the log.

8.18 event_commands.cpp

Go to the documentation of this file.

```
#include "communication.h"
#include "event_manager.h"
#include <sstream>


```

```
std::vector<Frame> handle_get_last_events(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType != OperationType::GET) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
        return frames;
    }

    size_t count = 10; // Default number of events to return
    if (!param.empty()) {
        try {
            count = std::stoul(param);
            if (count > EVENT_BUFFER_SIZE) {
                error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
                frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
                return frames;
            }
        } catch (...) {
            error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
            frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
            return frames;
        }
    }

    auto& event_manager = EventManager::get_instance();
    size_t available = event_manager.get_event_count();
    size_t to_return = (count == 0) ? available : std::min(count, available);
    size_t event_index = available;

    while (to_return > 0) {
        std::stringstream ss;
        ss « std::hex « std::uppercase « std::setfill('0');
        size_t events_in_frame = 0;

        for (size_t i = 0; i < 10 && to_return > 0; ++i) {
            event_index--;
            const EventLog& event = event_manager.get_event(event_index);

            ss « std::setw(4) « event.id
               « std::setw(8) « event.timestamp
               « std::setw(2) « static_cast<int>(event.group)
               « std::setw(2) « static_cast<int>(event.event);

            if (to_return > 1) ss « "-";
            to_return--;
            events_in_frame++;
        }
        frames.push_back(frame_build(OperationType::SEQ, 5, 1, ss.str()));
    }
    frames.push_back(frame_build(OperationType::VAL, 5, 1, "SEQ_DONE"));
    return frames;
}


std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType != OperationType::GET || !param.empty()) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, 5, 2, error_msg));
        return frames;
    }

    auto& event_manager = EventManager::get_instance();
    frames.push_back(frame_build(OperationType::VAL, 5, 2,
                        std::to_string(event_manager.get_event_count())));
    return frames;
} // end of EventCommands group
```

8.19 lib/comms/commands/gps_commands.cpp File Reference

```
#include "communication.h"
#include "lib/location/gps_collector.h"
#include <sstream>
#include "system_state_manager.h"
```

Macros

- #define GPS_GROUP 7
- #define POWER_STATUS_COMMAND 1
- #define PASSTHROUGH_COMMAND 2
- #define RMC_DATA_COMMAND 3
- #define GGA_DATA_COMMAND 4

Functions

- std::vector< Frame > handle_gps_power_status (const std::string &param, OperationType operationType)
  Handler for controlling GPS module power state.
- std::vector< Frame > handle_enable_gps_uart_passthrough (const std::string &param, OperationType operationType)
  Handler for enabling GPS transparent mode (UART pass-through)
- std::vector< Frame > handle_get_rmc_data (const std::string &param, OperationType operationType)
  Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.
- std::vector< Frame > handle_get_gga_data (const std::string &param, OperationType operationType)
  Handler for retrieving GPS GGA (Global Positioning System Fix Data) data.

8.19.1 Macro Definition Documentation

8.19.1.1 GPS_GROUP

```
#define GPS_GROUP 7
```

Definition at line 6 of file gps_commands.cpp.

8.19.1.2 POWER_STATUS_COMMAND

```
#define POWER_STATUS_COMMAND 1
```

Definition at line 7 of file gps_commands.cpp.

8.19.1.3 PASSTHROUGH_COMMAND

```
#define PASSTHROUGH_COMMAND 2
```

Definition at line 8 of file gps_commands.cpp.

8.19.1.4 RMC_DATA_COMMAND

```
#define RMC_DATA_COMMAND 3
```

Definition at line 9 of file gps_commands.cpp.

8.19.1.5 GGA_DATA_COMMAND

```
#define GGA_DATA_COMMAND 4
```

Definition at line 10 of file gps_commands.cpp.

8.20 gps_commands.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002 #include "lib/location/gps_collector.h"
00003 #include <sstream>
00004 #include "system_state_manager.h"
00005
00006 #define GPS_GROUP 7
00007 #define POWER_STATUS_COMMAND 1
00008 #define PASSTHROUGH_COMMAND 2
00009 #define RMC_DATA_COMMAND 3
00010 #define GGA_DATA_COMMAND 4
00011
00017
00033 std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType) {
00034     std::vector<Frame> frames;
00035     std::string error_str;
00036
00037     if (operationType == OperationType::SET) {
00038         if (param.empty()) {
00039             error_str = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00040             frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
      error_str));
00041             return frames;
00042         }
00043
00044         try {
00045             int power_status = std::stoi(param);
00046             if (power_status != 0 && power_status != 1) {
00047                 error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
00048                 frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
      error_str));
00049                 return frames;
00050             }
00051             gpio_put(GPS_POWER_ENABLE_PIN, power_status);
00052             EventEmitter::emit(EventGroup::GPS, power_status ? GPSEvent::POWER_ON :
      GPSEvent::POWER_OFF);
00053             frames.push_back(frame_build(OperationType::RES, GPS_GROUP, POWER_STATUS_COMMAND,
      std::to_string(power_status)));
00054             return frames;
00055         } catch (...) {
00056             error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
00057             frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
      error_str));
00058             return frames;
00059         }
00060     }
00061
00062     // GET operation
00063     if (!param.empty()) {
00064         error_str = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00065         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
00066         return frames;
00067     }
00068
00069     bool power_status = gpio_get(GPS_POWER_ENABLE_PIN);
00070     frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, POWER_STATUS_COMMAND,
      std::to_string(power_status)));
00071     return frames;
00072 }
00073
00074
00090 std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType
      operationType) {
00091     std::vector<Frame> frames;
00092     std::string error_str;
00093
00094     if (!(operationType == OperationType::SET)) {
00095         error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
00096         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
00097         return frames;
```

```
    }

    // Parse and validate timeout parameter
    uint32_t timeout_ms;
    try {
        timeout_ms = param.empty() ? 60000u : std::stoul(param) * 1000;
    } catch (...) {
        error_str = error_code_to_string(ErrorCode::INVALID_VALUE);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
        return frames;
    }

    // Setup UART parameters and exit sequence
    const std::string EXIT_SEQUENCE = "##EXIT##";
    std::string input_buffer;
    bool exit_requested = false;
    SystemStateManager::get_instance().set_gps_collection_paused(true);
    sleep_ms(100);

    uint32_t original_baud_rate = DEBUG_UART_BAUD_RATE;
    uint32_t gps_baud_rate = GPS_UART_BAUD_RATE;
    uint32_t start_time = to_ms_since_boot(get_absolute_time());

    EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_START);

    std::string message = "Entering GPS Serial Pass-Through Mode @" +
                          std::to_string(gps_baud_rate) + " for " +
                          std::to_string(timeout_ms/1000) + "s\r\n" +
                          "Send " + EXIT_SEQUENCE + " to exit";
    uart_print(message, VerbosityLevel::INFO);

    sleep_ms(10);

    // Change main UART baudrate to GPS module baudrate for passthrough duration
    uart_set_baudrate(DEBUG_UART_PORT, gps_baud_rate);

    while (!exit_requested) {
        while (uart_is_readable(DEBUG_UART_PORT)) {
            char ch = uart_getc(DEBUG_UART_PORT);

            input_buffer += ch;
            if (input_buffer.length() > EXIT_SEQUENCE.length()) {
                input_buffer = input_buffer.substr(1);
            }

            if (input_buffer == EXIT_SEQUENCE) {
                exit_requested = true;
                break;
            }

            if (input_buffer != EXIT_SEQUENCE.substr(0, input_buffer.length())) {
                uart_write_blocking(GPS_UART_PORT,
                    reinterpret_cast<const uint8_t*>(&ch), 1);
            }
        }

        while (uart_is_readable(GPS_UART_PORT)) {
            char gps_byte = uart_getc(GPS_UART_PORT);
            uart_write_blocking(DEBUG_UART_PORT,
                reinterpret_cast<const uint8_t*>(&gps_byte), 1);
        }

        if (to_ms_since_boot(get_absolute_time()) - start_time >= timeout_ms) {
            break;
        }
    }

    uart_set_baudrate(DEBUG_UART_PORT, original_baud_rate);

    sleep_ms(50);

    SystemStateManager::get_instance().set_gps_collection_paused(false);
    EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_END);

    std::string exit_reason = exit_requested ? "USER_EXIT" : "TIMEOUT";
    std::string response = "GPS UART BRIDGE EXIT: " + exit_reason;
    uart_print(response, VerbosityLevel::INFO);

    frames.push_back(frame_build(OperationType::RES, GPS_GROUP, PASSTHROUGH_COMMAND, response));
    return frames;
}


std::vector<Frame> handle_get_rmc_data(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

```

```
    if (operationType != OperationType::GET) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, RMC_DATA_COMMAND, error_msg));
        return frames;
    }

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, RMC_DATA_COMMAND, error_msg));
        return frames;
    }

    auto& nmea_data = NMEAData::get_instance();
    std::vector<std::string> tokens = nmea_data.get_rmc_tokens();
    std::stringstream ss;
    for (size_t i = 0; i < tokens.size(); ++i) {
        ss « tokens[i];
        if (i < tokens.size() - 1) {
            ss « ",";
        }
    }

    frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, RMC_DATA_COMMAND, ss.str()));
    return frames;
}


std::vector<Frame> handle_get_gga_data(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType != OperationType::GET) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, GGA_DATA_COMMAND, error_msg));
        return frames;
    }

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, GGA_DATA_COMMAND, error_msg));
        return frames;
    }

    auto& nmea_data = NMEAData::get_instance();
    std::vector<std::string> tokens = nmea_data.get_gga_tokens();
    std::stringstream ss;
    for (size_t i = 0; i < tokens.size(); ++i) {
        ss « tokens[i];
        if (i < tokens.size() - 1) {
            ss « ",";
        }
    }

    frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, GGA_DATA_COMMAND, ss.str()));
    return frames;
} // end of GPSCommands group
```

8.21 lib/comms/commands/power_commands.cpp File Reference

```
#include "communication.h"
```

Macros

- #define POWER_GROUP 2
- #define POWER_MANAGER_IDS 0
- #define VOLTAGE_BATTERY 2
- #define VOLTAGE_MAIN 3
- #define CHARGE_USB 4
- #define CHARGE_SOLAR 5
- #define CHARGE_TOTAL 6
- #define DRAW_TOTAL 7

Functions

- std::vector< Frame > handle_get_power_manager_ids (const std::string &param, OperationType operationType)

  Handler for retrieving Power Manager IDs.
- std::vector< Frame > handle_get_voltage_battery (const std::string &param, OperationType operationType)

  Handler for getting battery voltage.
- std::vector< Frame > handle_get_voltage_5v (const std::string &param, OperationType operationType)

  Handler for getting 5V rail voltage.
- std::vector< Frame > handle_get_current_charge_usb (const std::string &param, OperationType operation↩Type)

  Handler for getting USB charge current.
- std::vector< Frame > handle_get_current_charge_solar (const std::string &param, OperationType operationType)

  Handler for getting solar panel charge current.
- std::vector< Frame > handle_get_current_charge_total (const std::string &param, OperationType operationType)

  Handler for getting total charge current.
- std::vector< Frame > handle_get_current_draw (const std::string &param, OperationType operationType)

  Handler for getting system current draw.

8.21.1 Macro Definition Documentation

8.21.1.1 POWER_GROUP

```
#define POWER_GROUP 2
```

Definition at line 3 of file power_commands.cpp.

8.21.1.2 POWER_MANAGER_IDS

```
#define POWER_MANAGER_IDS 0
```

Definition at line 4 of file power_commands.cpp.

8.21.1.3 VOLTAGE_BATTERY

```
#define VOLTAGE_BATTERY 2
```

Definition at line 5 of file power_commands.cpp.

8.21.1.4 VOLTAGE_MAIN

```
#define VOLTAGE_MAIN 3
```

Definition at line 6 of file power_commands.cpp.

8.21.1.5 CHARGE_USB

```
#define CHARGE_USB 4
```

Definition at line 7 of file power_commands.cpp.

8.21.1.6 CHARGE_SOLAR

```
#define CHARGE_SOLAR 5
```

Definition at line 8 of file power_commands.cpp.

8.21.1.7 CHARGE_TOTAL

```
#define CHARGE_TOTAL 6
```

Definition at line 9 of file power_commands.cpp.

8.21.1.8 DRAW_TOTAL

```
#define DRAW_TOTAL 7
```

Definition at line 10 of file power_commands.cpp.

8.22 power_commands.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002
00003 #define POWER_GROUP 2
00004 #define POWER_MANAGER_IDS 0
00005 #define VOLTAGE_BATTERY 2
00006 #define VOLTAGE_MAIN 3
00007 #define CHARGE_USB 4
00008 #define CHARGE_SOLAR 5
00009 #define CHARGE_TOTAL 6
00010 #define DRAW_TOTAL 7
00011
00017
00030 std::vector<Frame> handle_get_power_manager_ids(const std::string& param, OperationType operationType)
      {
00031     std::vector<Frame> frames;
00032     std::string error_msg;
00033
00034     if (!param.empty()) {
00035         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00036         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, POWER_MANAGER_IDS, error_msg));
00037         return frames;
00038     }
00039
00040     if (!(operationType == OperationType::GET)) {
00041         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00042         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, POWER_MANAGER_IDS, error_msg));
00043         return frames;
00044     }
00045
00046     std::string power_manager_ids = PowerManager::get_instance().read_device_ids();
00047     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, POWER_MANAGER_IDS,
      power_manager_ids));
00048     return frames;
00049 }
```

```

std::vector<Frame> handle_get_voltage_battery(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_BATTERY, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_BATTERY, error_msg));
        return frames;
    }

    float voltage = PowerManager::get_instance().get_voltage_battery();
    frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, VOLTAGE_BATTERY, std::to_string(voltage), ValueUnit::VOLT));
    return frames;
}

std::vector<Frame> handle_get_voltage_5v(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_MAIN, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_MAIN, error_msg));
        return frames;
    }

    float voltage = PowerManager::get_instance().get_voltage_5v();
    frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, VOLTAGE_MAIN, std::to_string(voltage), ValueUnit::VOLT));
    return frames;
}

std::vector<Frame> handle_get_current_charge_usb(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_USB, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_USB, error_msg));
        return frames;
    }

    float charge_current = PowerManager::get_instance().get_current_charge_usb();
    frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_USB, std::to_string(charge_current), ValueUnit::MILIAMP));
    return frames;
}

std::vector<Frame> handle_get_current_charge_solar(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_SOLAR, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_SOLAR, error_msg));
        return frames;
    }

    float charge_current = PowerManager::get_instance().get_current_charge_solar();
    frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_SOLAR,
```

```
std::to_string(charge_current), ValueUnit::MILIAMP));
    return frames;
}

std::vector<Frame> handle_get_current_charge_total(const std::string& param, OperationType
operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_TOTAL, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_TOTAL, error_msg));
        return frames;
    }

    float charge_current = PowerManager::get_instance().get_current_charge_total();
    frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_TOTAL,
std::to_string(charge_current), ValueUnit::MILIAMP));
    return frames;
}

std::vector<Frame> handle_get_current_draw(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (!param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, DRAW_TOTAL, error_msg));
        return frames;
    }

    if (!(operationType == OperationType::GET)) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, DRAW_TOTAL, error_msg));
        return frames;
    }

    float current_draw = PowerManager::get_instance().get_current_draw();
    frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, DRAW_TOTAL,
std::to_string(current_draw), ValueUnit::MILIAMP));
    return frames;
} // end of PowerCommands group
```

8.23 lib/comms/commands/sensor_commands.cpp File Reference

```
#include "communication.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <sstream>
#include "commands.h"
```

Macros

- #define SENSOR_GROUP 4
- #define SENSOR_READ 0
- #define SENSOR_CONFIGURE 1

Functions

- std::vector< Frame > handle_get_sensor_data (const std::string &param, OperationType operationType)

    Handler for reading sensor data.

- std::vector< Frame > handle_sensor_config (const std::string &param, OperationType operationType)

    Handler for configuring sensors.
- std::vector< Frame > handle_get_sensor_list (const std::string &param, OperationType operationType)

    Handler for listing available sensors.

8.23.1 Macro Definition Documentation

8.23.1.1 SENSOR_GROUP

```
#define SENSOR_GROUP 4
```

Definition at line 8 of file sensor_commands.cpp.

8.23.1.2 SENSOR_READ

```
#define SENSOR_READ 0
```

Definition at line 9 of file sensor_commands.cpp.

8.23.1.3 SENSOR_CONFIGURE

```
#define SENSOR_CONFIGURE 1
```

Definition at line 10 of file sensor_commands.cpp.

8.24 sensor_commands.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002 #include "ISensor.h"
00003 #include <vector>
00004 #include <string>
00005 #include <sstream>
00006 #include "commands.h"
00007
00008 #define SENSOR_GROUP 4
00009 #define SENSOR_READ 0
00010 #define SENSOR_CONFIGURE 1
00011
00017
00034 std::vector<Frame> handle_get_sensor_data(const std::string& param, OperationType operationType) {
00035     std::vector<Frame> frames;
00036     std::string error_msg;
00037
00038     if (param.empty()) {
00039         error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00040         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
00041         return frames;
00042     }
00043
00044     if (operationType != OperationType::GET) {
00045         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00046         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
00047         return frames;
00048     }
00049
00050     // Parse sensor type and data type from param
00051     std::string sensor_type_str;
00052     std::string data_type_str;
00053
```

```
    size_t dash_pos = param.find('-');
    if (dash_pos != std::string::npos) {
        sensor_type_str = param.substr(0, dash_pos);
        data_type_str = param.substr(dash_pos + 1);
    } else {
        sensor_type_str = param;
    }

    // Convert sensor_type_str to SensorType
    SensorType sensor_type;
    if (sensor_type_str == "light") {
        sensor_type = SensorType::LIGHT;
    } else if (sensor_type_str == "environment") {
        sensor_type = SensorType::ENVIRONMENT;
    } else {
        error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid sensor type";
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
        return frames;
    }

    SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();

    // If data type is specified, read that specific data
    if (!data_type_str.empty()) {
        SensorDataTypeIdentifier data_type;

        // Map string to SensorDataTypeIdentifier
        if (data_type_str == "light_level") {
            data_type = SensorDataTypeIdentifier::LIGHT_LEVEL;
        } else if (data_type_str == "temperature") {
            data_type = SensorDataTypeIdentifier::TEMPERATURE;
        } else if (data_type_str == "pressure") {
            data_type = SensorDataTypeIdentifier::PRESSURE;
        } else if (data_type_str == "humidity") {
            data_type = SensorDataTypeIdentifier::HUMIDITY;
        } else {
            error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid data type";
            frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
            return frames;
        }

        float value = sensor_wrapper.sensor_read_data(sensor_type, data_type);
        std::stringstream ss;
        ss « value;
        frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, SENSOR_READ, ss.str()));
    }
    // If only sensor type is specified, read all relevant data for that sensor type
    else {
        std::vector<SensorDataTypeIdentifier> data_types;

        switch (sensor_type) {
            case SensorType::LIGHT:
                data_types = {SensorDataTypeIdentifier::LIGHT_LEVEL};
                break;
            case SensorType::ENVIRONMENT:
                data_types = {
                    SensorDataTypeIdentifier::TEMPERATURE,
                    SensorDataTypeIdentifier::PRESSURE,
                    SensorDataTypeIdentifier::HUMIDITY
                };
                break;
            default:
                break;
        }

        std::stringstream combined_values;
        std::vector<std::string> data_type_names;
        std::vector<float> values;

        // Get names for the data types and store the values
        for (SensorDataTypeIdentifier data_type : data_types) {
            switch (data_type) {
                case SensorDataTypeIdentifier::LIGHT_LEVEL:
                    data_type_names.push_back("light_level");
                    break;
                case SensorDataTypeIdentifier::TEMPERATURE:
                    data_type_names.push_back("temperature");
                    break;
                case SensorDataTypeIdentifier::PRESSURE:
                    data_type_names.push_back("pressure");
                    break;
                case SensorDataTypeIdentifier::HUMIDITY:
                    data_type_names.push_back("humidity");
                    break;
                default:
                    break;
            }
```

```
            float value = sensor_wrapper.sensor_read_data(sensor_type, data_type);
            values.push_back(value);
        }

        // Format output as key-value pairs
        for (size_t i = 0; i < data_type_names.size(); i++) {
            if (i > 0) combined_values « "|";
            combined_values « data_type_names[i] « ":" « values[i];
        }

        frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, SENSOR_READ,
combined_values.str()));
    }

    return frames;
}

std::vector<Frame> handle_sensor_config(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (param.empty()) {
        error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
        return frames;
    }

    if (operationType != OperationType::SET) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
        return frames;
    }

    // Parse sensor type and configuration from param
    size_t semicolon_pos = param.find(';');
    if (semicolon_pos == std::string::npos) {
        error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Format should be
sensor_type;config_params";
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
        return frames;
    }

    std::string sensor_type_str = param.substr(0, semicolon_pos);
    std::string config_str = param.substr(semicolon_pos + 1);

    // Convert sensor_type_str to SensorType
    SensorType sensor_type;
    if (sensor_type_str == "light") {
        sensor_type = SensorType::LIGHT;
    } else if (sensor_type_str == "environment") {
        sensor_type = SensorType::ENVIRONMENT;
    } else {
        error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid sensor type";
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
        return frames;
    }

    // Parse configuration parameters
    std::map<std::string, std::string> config_map;
    std::stringstream ss(config_str);
    std::string config_pair;

    while (std::getline(ss, config_pair, '|')) {
        size_t colon_pos = config_pair.find(':');
        if (colon_pos != std::string::npos) {
            std::string key = config_pair.substr(0, colon_pos);
            std::string value = config_pair.substr(colon_pos + 1);
            config_map[key] = value;
        }
    }

    // Apply configuration
    SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
    bool success = sensor_wrapper.sensor_configure(sensor_type, config_map);

    if (success) {
        frames.push_back(frame_build(OperationType::RES, SENSOR_GROUP, SENSOR_CONFIGURE,
"Configuration successful"));
    } else {
        error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET) + ": Failed to configure sensor";
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
    }

    return frames;
}
```

```
std::vector<Frame> handle_get_sensor_list([[maybe_unused]] const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType != OperationType::GET) {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, 2, error_msg));
        return frames;
    }

    // Get the singleton instance
    SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();

    // Get list of available sensor types
    std::vector<std::pair<SensorType, uint8_t» available_sensors = sensor_wrapper.get_available_sensors();

    if (available_sensors.empty()) {
        frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, 2, "No sensors available"));
        return frames;
    }

    std::stringstream sensor_list;
    bool first = true;

    for (const auto& sensor_info : available_sensors) {
        if (!first) {
            sensor_list « "|";
        }

        // Format: sensor_type:address (in hex)
        std::stringstream addr_hex;
        addr_hex « std::hex « static_cast<int>(sensor_info.second);

        switch (sensor_info.first) {
            case SensorType::LIGHT:
                sensor_list « "light:0x" « addr_hex.str();
                break;
            case SensorType::ENVIRONMENT:
                sensor_list « "environment:0x" « addr_hex.str();
                break;
            default:
                sensor_list « "unknown:0x" « addr_hex.str();
                break;
        }

        first = false;
    }

    frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, 2, sensor_list.str()));
    return frames;
}
```

8.25 lib/comms/commands/storage_commands.cpp File Reference

```
#include "commands.h"
#include "communication.h"
#include "storage.h"
#include "filesystem/vfs.h"
#include "filesystem/littlefs.h"
#include <sys/stat.h>
#include <errno.h>
#include "dirent.h"
```

Macros

- #define STORAGE_GROUP 6
- #define LIST_FILES_COMMAND 0
- #define MOUNT_COMMAND 4

Functions

- std::vector< Frame > handle_list_files (const std::string &param, OperationType operationType)

  Handles the list files command.
- std::vector< Frame > handle_mount (const std::string &param, OperationType operationType)

  Handles the SD card mount/unmount command.

8.25.1 Macro Definition Documentation

8.25.1.1 STORAGE_GROUP

```
#define STORAGE_GROUP 6
```

Definition at line 10 of file storage_commands.cpp.

8.25.1.2 LIST_FILES_COMMAND

```
#define LIST_FILES_COMMAND 0
```

Definition at line 12 of file storage_commands.cpp.

8.25.1.3 MOUNT_COMMAND

```
#define MOUNT_COMMAND 4
```

Definition at line 13 of file storage_commands.cpp.

8.26 storage_commands.cpp

Go to the documentation of this file.

```
00001 #include "commands.h"
00002 #include "communication.h"
00003 #include "storage.h"
00004 #include "filesystem/vfs.h"
00005 #include "filesystem/littlefs.h"
00006 #include <sys/stat.h>
00007 #include <errno.h>
00008 #include "dirent.h"
00009
00010 #define STORAGE_GROUP 6
00011
00012 #define LIST_FILES_COMMAND 0
00013 #define MOUNT_COMMAND 4
00019
00037 std::vector<Frame> handle_list_files([[maybe_unused]] const std::string& param, OperationType
      operationType) {
00038     std::vector<Frame> frames;
00039     std::string error_msg;
00040
00041     if (operationType != OperationType::GET) {
00042         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00043         frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND,
      error_msg));
00044         return frames;
00045     }
00046
00047     DIR* dir;
00048     struct dirent* ent;
00049     int file_count = 0;
```

```
    if ((dir = opendir("/")) != NULL) {
        // First, count the number of files
        while ((ent = readdir(dir)) != NULL) {
            const char* filename = ent->d_name;
            if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
                continue;
            }
            file_count++;
        }
        closedir(dir);

        // Send the number of files
        frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, LIST_FILES_COMMAND, std::to_string(file_count)));

        // Open the directory again to read file information
        dir = opendir("/");
        if (dir != NULL) {
            while ((ent = readdir(dir)) != NULL) {
                const char* filename = ent->d_name;

                // Skip "." and ".." directories
                if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
                    continue;
                }

                // Get file size
                char filepath[256];
                int written = snprintf(filepath, sizeof(filepath), "/%s", filename);
                if (written < 0 || written >= static_cast<int>(sizeof(filepath))) {
                    continue; // Skip this file if path is too long
                }

                FILE* file = fopen(filepath, "rb");
                size_t file_size = 0;

                if (file != NULL) {
                    fseek(file, 0, SEEK_END);
                    file_size = ftell(file);
                    fclose(file);
                }

                // Create and send frame with filename and size
                char file_info[512];
                snprintf(file_info, sizeof(file_info), "%s:%zu", filename, file_size);
                frames.push_back(frame_build(OperationType::SEQ, STORAGE_GROUP, LIST_FILES_COMMAND, file_info));
            }
            closedir(dir);
            frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, LIST_FILES_COMMAND, "SEQ_DONE"));
            return frames;
        } else {
            error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
            frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND, error_msg));
            return frames;
        }
    } else {
        frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND, error_msg));
        return frames;
    }
}

std::vector<Frame> handle_mount(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_msg;

    if (operationType == OperationType::GET) {
        bool state = SystemStateManager::get_instance().is_sd_card_mounted();

        frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, MOUNT_COMMAND, std::to_string(state)));
        return frames;
    } else if (operationType == OperationType::SET) {
        if (param == "1") {
            if (fs_init()) {
                frames.push_back(frame_build(OperationType::RES, STORAGE_GROUP, MOUNT_COMMAND, "SD_MOUNT_OK"));
                return frames;
            } else {
                error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
                frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND, error_msg));
                return frames;
            }
```

```
        } else if (param == "0") {
            if (fs_unmount("/") == 0) {
                if (SystemStateManager::get_instance().is_sd_card_mounted()) {
                    frames.push_back(frame_build(OperationType::RES, STORAGE_GROUP, MOUNT_COMMAND, "SD_UNMOUNT_OK"));
                }
                return frames;
            } else {
                error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
                frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND, error_msg));
                return frames;
            }
        } else {
            error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
            frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND, error_msg));
            return frames;
        }
    } else {
        error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND, error_msg));
        return frames;
    }
} // StorageCommands
```

8.27 lib/comms/commands/telemetry_commands.cpp File Reference

```
#include "commands.h"
#include "communication.h"
#include "telemetry_manager.h"
```

Macros

- #define TELEMETRY_GROUP 8
- #define GET_LAST_TELEMETRY_RECORD_COMMAND 2
- #define GET_LAST_SENSOR_RECORD_COMMAND 3

Functions

- std::vector< Frame > handle_get_last_telemetry_record (const std::string &param, OperationType operationType)

  Handles the get last record command.
- std::vector< Frame > handle_get_last_sensor_record (const std::string &param, OperationType operationType)

  Handles the get last sensor record command.

8.27.1 Macro Definition Documentation

8.27.1.1 TELEMETRY_GROUP

```
#define TELEMETRY_GROUP 8
```

Definition at line 5 of file telemetry_commands.cpp.

8.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND

```
#define GET_LAST_TELEMETRY_RECORD_COMMAND 2
```

Definition at line 6 of file telemetry_commands.cpp.

8.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND

```
#define GET_LAST_SENSOR_RECORD_COMMAND 3
```

Definition at line 7 of file telemetry_commands.cpp.

8.28 telemetry_commands.cpp

Go to the documentation of this file.

```
00001 #include "commands.h"
00002 #include "communication.h"
00003 #include "telemetry_manager.h"
00004
00005 #define TELEMETRY_GROUP 8
00006 #define GET_LAST_TELEMETRY_RECORD_COMMAND 2
00007 #define GET_LAST_SENSOR_RECORD_COMMAND 3
00008
00014
00032 std::vector<Frame> handle_get_last_telemetry_record([[maybe_unused]] const std::string& param, OperationType operationType) {
00033     std::vector<Frame> frames;
00034     std::string error_msg;
00035
00036     if (operationType != OperationType::GET) {
00037         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00038         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP, GET_LAST_TELEMETRY_RECORD_COMMAND, error_msg));
00039         return frames;
00040     }
00041
00042     std::string csv_data = TelemetryManager::get_instance().get_last_telemetry_record_csv();
00043
00044     if (csv_data.empty()) {
00045         error_msg = "NO_DATA";
00046         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00047         return frames;
00048     }
00049     frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP, GET_LAST_TELEMETRY_RECORD_COMMAND, csv_data));
00050
00051     return frames;
00052 }
00053
00054
00065 std::vector<Frame> handle_get_last_sensor_record([[maybe_unused]]const std::string& param, OperationType operationType) {
00066     std::vector<Frame> frames;
00067     std::string error_msg;
00068
00069     if (operationType != OperationType::GET) {
00070         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00071         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00072         return frames;
00073     }
00074
00075     std::string csv_data = TelemetryManager::get_instance().get_last_sensor_record_csv();
00076
00077     if (csv_data.empty()) {
00078         error_msg = "NO_DATA";
00079         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00080         return frames;
00081     }
00082
00083     frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, csv_data));
00084
00085     return frames;
00086 } // TelemetryBufferCommands
```

8.29 lib/comms/communication.cpp File Reference

```
#include "communication.h"
```

Functions

- bool initialize_radio ()
  Initializes the LoRa radio module.
- void lora_tx_done_callback ()
  Callback function for LoRa transmission completion.

Variables

- string outgoing
- uint8_t msgCount = 0
- long lastSendTime = 0
- long lastReceiveTime = 0
- long lastPrintTime = 0
- unsigned long interval = 0

8.29.1 Function Documentation

8.29.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

Returns

True if initialization was successful, false otherwise.

Sets the LoRa pins and attempts to begin LoRa communication at a specified frequency. Emits a CommsEvent::RADIO_INIT event on success or a CommsEvent::RADIO_ERROR event on failure.

Definition at line 16 of file communication.cpp.

8.29.1.2 lora_tx_done_callback()

```
void lora_tx_done_callback ()
```

Callback function for LoRa transmission completion.

Prints a debug message to the UART and sets the LoRa module to receive mode.

Definition at line 44 of file communication.cpp.

8.29.2 Variable Documentation

8.29.2.1 outgoing

```
string outgoing
```

Definition at line 3 of file communication.cpp.

8.29.2.2 msgCount

```
uint8_t msgCount = 0
```

Definition at line 4 of file communication.cpp.

8.29.2.3 lastSendTime

```
long lastSendTime = 0
```

Definition at line 5 of file communication.cpp.

8.29.2.4 lastReceiveTime

```
long lastReceiveTime = 0
```

Definition at line 6 of file communication.cpp.

8.29.2.5 lastPrintTime

```
long lastPrintTime = 0
```

Definition at line 7 of file communication.cpp.

8.29.2.6 interval

```
unsigned long interval = 0
```

Definition at line 8 of file communication.cpp.

8.30 communication.cpp

Go to the documentation of this file.

```
#include "communication.h"

string outgoing;
uint8_t msgCount = 0;
long lastSendTime = 0;
long lastReceiveTime = 0;
long lastPrintTime = 0;
unsigned long interval = 0;

bool initialize_radio() {
    LoRa.set_pins(lora_cs_pin, lora_reset_pin, lora_irq_pin);
    long frequency = 433E6;
    bool init_status = false;
    if (!LoRa.begin(frequency))
    {
        uart_print("LoRa init failed. Check your connections.", VerbosityLevel::WARNING);
        init_status = false;
    } else {
        uart_print("LoRa initialized with frequency " + std::to_string(frequency),
VerbosityLevel::INFO);

        // Set up TxDone callback to automatically return to receive mode
        LoRa.onTxDone(lora_tx_done_callback);

        LoRa.receive(0);

        init_status = true;
    }

    EventEmitter::emit(EventGroup::COMMS, init_status ? CommsEvent::RADIO_INIT :
CommsEvent::RADIO_ERROR);

    return init_status;
}

void lora_tx_done_callback() {
    uart_print("LoRa transmission complete", VerbosityLevel::DEBUG);
    LoRa.receive(0);
}
```

8.31 lib/comms/communication.h File Reference

```
#include <string>
#include <vector>
#include "protocol.h"
#include "event_manager.h"
```

Functions

- bool initialize_radio ()
  Initializes the LoRa radio module.
- void lora_tx_done_callback ()
  Callback function for LoRa transmission completion.
- void on_receive (int packetSize)
  Callback function for handling received LoRa packets.
- void handle_uart_input ()
  Handles UART input.
- void send_message (std::string outgoing)
- void send_frame_uart (const Frame &frame)
- void send_frame_lora (const Frame &frame)
- void split_and_send_message (const uint8_t *data, size_t length)

- std::vector< Frame > execute_command (uint32_t commandKey, const std::string &param, OperationType operationType)

    Executes a command based on its key.
- void frame_process (const std::string &data, Interface interface)

    Executes a command based on the command key and the parameter.
- std::string frame_encode (const Frame &frame)

    Encodes a Frame instance into a string.
- Frame frame_decode (const std::string &data)

    Decodes a string into a Frame instance.
- Frame frame_build (OperationType operation, uint8_t group, uint8_t command, const std::string &value, const ValueUnit unitType=ValueUnit::UNDEFINED)

    Builds a Frame instance based on the execution result, group, command, value, and unit.
- std::string determine_unit (uint8_t group, uint8_t command)

8.31.1 Function Documentation

8.31.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

Returns

True if initialization was successful, false otherwise.

Sets the LoRa pins and attempts to begin LoRa communication at a specified frequency. Emits a CommsEvent::RADIO_INIT event on success or a CommsEvent::RADIO_ERROR event on failure.

Definition at line 16 of file communication.cpp.

8.31.1.2 lora_tx_done_callback()

```
void lora_tx_done_callback ()
```

Callback function for LoRa transmission completion.

Prints a debug message to the UART and sets the LoRa module to receive mode.

Definition at line 44 of file communication.cpp.

8.31.1.3 send_message()

```
void send_message (
            std::string outgoing)
```

8.31.1.4 send_frame_uart()

```
void send_frame_uart (
            const Frame & frame)
```

Definition at line 47 of file send.cpp.

8.31.1.5 send_frame_lora()

```
void send_frame_lora (
            const Frame & frame)
```

Definition at line 39 of file send.cpp.

8.31.1.6 split_and_send_message()

```
void split_and_send_message (
            const uint8_t * data,
            size_t length)
```

8.31.1.7 determine_unit()

```
std::string determine_unit (
            uint8_t group,
            uint8_t command)
```

8.32 communication.h

Go to the documentation of this file.

```
#ifndef COMMUNICATION_H
#define COMMUNICATION_H

#include <string>
#include <vector>
#include "protocol.h"
#include "event_manager.h"

bool initialize_radio();
void lora_tx_done_callback();
void on_receive(int packetSize);
void handle_uart_input();
void send_message(std::string outgoing);
void send_frame_uart(const Frame& frame);
void send_frame_lora(const Frame& frame);

void split_and_send_message(const uint8_t* data, size_t length);

std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
    operationType);

void frame_process(const std::string& data, Interface interface);
std::string frame_encode(const Frame& frame);
Frame frame_decode(const std::string& data);
Frame frame_build(OperationType operation, uint8_t group, uint8_t command,const std::string& value,
    const ValueUnit unitType  = ValueUnit::UNDEFINED);

std::string determine_unit(uint8_t group, uint8_t command);

#endif
```

8.33 lib/comms/frame.cpp File Reference

Implements functions for encoding, decoding, building, and processing Frames.

```
#include "communication.h"
```

Typedefs

- using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>

Functions

- std::string frame_encode (const Frame &frame)

    Encodes a Frame instance into a string.
- Frame frame_decode (const std::string &data)

    Decodes a string into a Frame instance.
- void frame_process (const std::string &data, Interface interface)

    Executes a command based on the command key and the parameter.
- Frame frame_build (OperationType operation, uint8_t group, uint8_t command, const std::string &value, const ValueUnit unitType)

    Builds a Frame instance based on the execution result, group, command, value, and unit.

Variables

- std::map< uint32_t, CommandHandler > command_handlers

    Global map of all command handlers.
- volatile uint16_t eventRegister

8.33.1 Detailed Description

Implements functions for encoding, decoding, building, and processing Frames.

Definition in file frame.cpp.

8.33.2 Typedef Documentation

8.33.2.1 CommandHandler

```
using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>
```

Definition at line 3 of file frame.cpp.

8.33.3 Variable Documentation

8.33.3.1 eventRegister

```
volatile uint16_t eventRegister [extern]
```

8.34 frame.cpp

Go to the documentation of this file.

```
#include "communication.h"

using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>;
extern std::map<uint32_t, CommandHandler> command_handlers;
extern volatile uint16_t eventRegister;


std::string frame_encode(const Frame& frame) {
    std::stringstream ss;
    ss « static_cast<int>(frame.direction) « DELIMITER
       « operation_type_to_string(frame.operationType) « DELIMITER
       « static_cast<int>(frame.group) « DELIMITER
       « static_cast<int>(frame.command) « DELIMITER
       « frame.value;

    if (!frame.unit.empty()) {
        ss « DELIMITER « frame.unit;
    }

    return FRAME_BEGIN + DELIMITER + ss.str() + DELIMITER + FRAME_END;
}


Frame frame_decode(const std::string& data) {
    try {
        Frame frame;
        std::stringstream ss(data);
        std::string token;

        std::getline(ss, token, DELIMITER);
        if (token != FRAME_BEGIN)
            throw std::runtime_error("Invalid frame header");
        frame.header = token;

        std::string decoded_frame_data;
        while (std::getline(ss, token, DELIMITER)) {
            if (token == FRAME_END) break;
            decoded_frame_data += token + DELIMITER;
        }
        if (!decoded_frame_data.empty()) {
            decoded_frame_data.pop_back();
        }

        std::stringstream frame_data_stream(decoded_frame_data);

        std::getline(frame_data_stream, token, DELIMITER);
        frame.direction = std::stoi(token);

        std::getline(frame_data_stream, token, DELIMITER);
        frame.operationType = string_to_operation_type(token);

        std::getline(frame_data_stream, token, DELIMITER);
        frame.group = std::stoi(token);

        std::getline(frame_data_stream, token, DELIMITER);
        frame.command = std::stoi(token);

        std::getline(frame_data_stream, token, DELIMITER);
        frame.value = token;

        std::getline(frame_data_stream, token, DELIMITER);
        frame.unit = token;

        return frame;
    } catch (const std::exception& e) {
        uart_print("Frame error: " + std::string(e.what()), VerbosityLevel::ERROR);
        Frame error_frame = frame_build(OperationType::ERR, 0, 0, e.what());
        return error_frame;
    }
}


void frame_process(const std::string& data, Interface interface) {
    gpio_put(PICO_DEFAULT_LED_PIN, 0);

    uart_print("Processing frame: " + data, VerbosityLevel::INFO);
    try {
        Frame frame = frame_decode(data);
        uint32_t command_key = (static_cast<uint32_t>(frame.group) « 8) | static_cast<uint32_t>(frame.command);

```

```
        std::vector<Frame> response_frames = execute_command(command_key, frame.value, frame.operationType);

        gpio_put(PICO_DEFAULT_LED_PIN, 1);

        // Send all responses through the same interface that received the command
        for (const auto& response_frame : response_frames) {
            if (interface == Interface::UART) {
                send_frame_uart(response_frame);
            } else if (interface == Interface::LORA) {
                send_frame_lora(response_frame);
                sleep_ms(25);
            }
        }
    } catch (const std::exception& e) {
        Frame error_frame = frame_build(OperationType::ERR, 0, 0, e.what());
        if (interface == Interface::UART) {
            send_frame_uart(error_frame);
        } else if (interface == Interface::LORA) {
            send_frame_lora(error_frame);
        }
    }
}

Frame frame_build(OperationType operation, uint8_t group, uint8_t command,
                const std::string& value, const ValueUnit unitType) {
    Frame frame;
    frame.header = FRAME_BEGIN;
    frame.footer = FRAME_END;

    switch (operation) {
        case OperationType::VAL:
            frame.direction = 1;
            frame.operationType = OperationType::VAL;
            frame.value = value;
            frame.unit = value_unit_type_to_string(unitType);
            break;

        case OperationType::ERR:
            frame.direction = 1;
            frame.operationType = OperationType::ERR;
            frame.value = value;
            frame.unit = value_unit_type_to_string(ValueUnit::UNDEFINED);
            break;

        case OperationType::RES:
            frame.direction = 1;
            frame.operationType = OperationType::RES;
            frame.value = value;
            frame.unit = value_unit_type_to_string(unitType);
            break;

        case OperationType::SEQ:
            frame.direction = 1;
            frame.operationType = OperationType::SEQ;
            frame.value = value;
            frame.unit = value_unit_type_to_string(unitType);
            break;
        default:
            break;
    }

    frame.group = group;
    frame.command = command;

    return frame;
}
 // end of FrameHandling group
```

8.35 lib/comms/protocol.h File Reference

```
#include <string>
#include <map>
#include <functional>
#include <vector>
#include <cstdint>
#include <iomanip>
#include "pin_config.h"
```

```
#include "PowerManager.h"
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include "utils.h"
#include "time.h"
#include "build_number.h"
#include "LoRa/LoRa-RP2040.h"
```

Classes

- struct Frame

  Represents a communication frame used for data exchange.

Enumerations

- enum class ErrorCode {
  ErrorCode::PARAM_UNNECESSARY , ErrorCode::PARAM_REQUIRED , ErrorCode::PARAM_INVALID , ErrorCode::INVALID_OPERATION ,
  ErrorCode::NOT_ALLOWED , ErrorCode::INVALID_FORMAT , ErrorCode::INVALID_VALUE , ErrorCode::FAIL_TO_SET
  ,
  ErrorCode::INTERNAL_FAIL_TO_READ , ErrorCode::UNKNOWN_ERROR }

  Standard error codes for command responses.
- enum class OperationType {
  OperationType::GET , OperationType::SET , OperationType::RES , OperationType::VAL ,
  OperationType::SEQ , OperationType::ERR }

  Represents the type of operation being performed.
- enum class CommandAccessLevel { CommandAccessLevel::NONE , CommandAccessLevel::READ_ONLY
  , CommandAccessLevel::WRITE_ONLY , CommandAccessLevel::READ_WRITE }

  Represents the access level required to execute a command.
- enum class ValueUnit {
  ValueUnit::UNDEFINED , ValueUnit::SECOND , ValueUnit::VOLT , ValueUnit::BOOL ,
  ValueUnit::DATETIME , ValueUnit::TEXT , ValueUnit::MILIAMP }

  Represents the unit of measurement for a payload value.
- enum class ExceptionType {
  ExceptionType::NONE , ExceptionType::NOT_ALLOWED , ExceptionType::INVALID_PARAM , ExceptionType::INVALID_OPER
  ,
  ExceptionType::PARAM_UNECESSARY }

  Represents the type of exception that occurred during command execution.
- enum class Interface { Interface::UART , Interface::LORA }

  Represents the communication interface being used.

Functions

- std::string exception_type_to_string (ExceptionType type)

  Converts an ExceptionType to a string.
- std::string error_code_to_string (ErrorCode code)

  Converts an ErrorCode to its string representation.
- std::string operation_type_to_string (OperationType type)

  Converts an OperationType to a string.
- OperationType string_to_operation_type (const std::string &str)

Converts a string to an OperationType.

- std::vector< uint8_t > hex_string_to_bytes (const std::string &hexString)

  Converts a hex string to a vector of bytes.

- std::string value_unit_type_to_string (ValueUnit unit)

  Converts a ValueUnit to a string.

Variables

- const std::string FRAME_BEGIN = "KBST"
- const std::string FRAME_END = "TSBK"
- const char DELIMITER = ';'

8.35.1 Variable Documentation

8.35.1.1 FRAME_BEGIN

```
const std::string FRAME_BEGIN = "KBST"
```

Definition at line 31 of file protocol.h.

8.35.1.2 FRAME_END

```
const std::string FRAME_END = "TSBK"
```

Definition at line 38 of file protocol.h.

8.35.1.3 DELIMITER

```
const char DELIMITER = ';'
```

Definition at line 45 of file protocol.h.

8.36 protocol.h

Go to the documentation of this file.

```
00001 // protocol.h
00002 #ifndef PROTOCOL_H
00003 #define PROTOCOL_H
00004
00005 #include <string>
00006 #include <map>
00007 #include <functional>
00008 #include <vector>
00009 #include <cstdint>
00010 #include <iomanip>
00011 #include "pin_config.h"
00012 #include "PowerManager.h"
00013 #include <cstdio>
00014 #include <cstdlib>
00015 #include <map>
00016 #include <cstring>
00017 #include "utils.h"
00018 #include "time.h"
00019 #include "build_number.h"
```

```cpp
#include "LoRa/LoRa-RP2040.h"

const std::string FRAME_BEGIN = "KBST";

const std::string FRAME_END = "TSBK";

const char DELIMITER = ';';


enum class ErrorCode {
    PARAM_UNNECESSARY,    // Parameter provided but not needed
    PARAM_REQUIRED,       // Required parameter missing
    PARAM_INVALID,        // Parameter has invalid format or value
    INVALID_OPERATION,    // Operation not allowed for this command
    NOT_ALLOWED,          // Operation not permitted
    INVALID_FORMAT,       // Input format is incorrect
    INVALID_VALUE,        // Value is outside expected range
    FAIL_TO_SET,          // Failed to set provided value
    INTERNAL_FAIL_TO_READ,// Failed to read from device in remote
    UNKNOWN_ERROR         // Generic error
};


enum class OperationType {
    GET,
    SET,
    RES,
    VAL,
    SEQ,
    ERR,

};



enum class CommandAccessLevel {
    NONE,
    READ_ONLY,
    WRITE_ONLY,
    READ_WRITE
};



enum class ValueUnit {
    UNDEFINED,
    SECOND,
    VOLT,
    BOOL,
    DATETIME,
    TEXT,
    MILIAMP,
};



enum class ExceptionType {
    NONE,
    NOT_ALLOWED,
    INVALID_PARAM,
    INVALID_OPERATION,
    PARAM_UNECESSARY
};



enum class Interface {
    UART,
    LORA
};


struct Frame {
    std::string header;              // Start marker
    uint8_t direction;               // 0 = ground->sat, 1 = sat->ground
    OperationType operationType;
    uint8_t group;                   // Group ID
    uint8_t command;                 // Command ID within group
    std::string value;               // Payload value
    std::string unit;                // Payload unit
    std::string footer;              // End marker
};

std::string exception_type_to_string(ExceptionType type);
std::string error_code_to_string(ErrorCode code);
std::string operation_type_to_string(OperationType type);
OperationType string_to_operation_type(const std::string& str);
```

```
std::vector<uint8_t> hex_string_to_bytes(const std::string& hexString);
std::string value_unit_type_to_string(ValueUnit unit);

#endif

```

8.37 lib/comms/receive.cpp File Reference

Implements functions for receiving and processing data, including LoRa and UART input.

```
#include "communication.h"
```

Macros

- #define MAX_PACKET_SIZE 255

Functions

- void on_receive (int packet_size)

  Callback function for handling received LoRa packets.
- void handle_uart_input ()

  Handles UART input.

8.37.1 Detailed Description

Implements functions for receiving and processing data, including LoRa and UART input.

Definition in file receive.cpp.

8.37.2 Macro Definition Documentation

8.37.2.1 MAX_PACKET_SIZE

```
#define MAX_PACKET_SIZE 255
```

Definition at line 3 of file receive.cpp.

8.38 receive.cpp

Go to the documentation of this file.

```
#include "communication.h"

#define MAX_PACKET_SIZE 255


void on_receive(int packet_size) {
    if (packet_size == 0) return;
    uart_print("Received LoRa packet of size " + std::to_string(packet_size), VerbosityLevel::DEBUG);

    std::vector<uint8_t> buffer;
    buffer.reserve(packet_size);

    int bytes_read = 0;

    while (LoRa.available() && bytes_read < packet_size) {
        if (bytes_read >= MAX_PACKET_SIZE) {
            uart_print("Error: Packet exceeds maximum allowed size!", VerbosityLevel::ERROR);
            return;
        }
        buffer.push_back(LoRa.read());
        bytes_read++;
    }

    if (bytes_read < 2) {
        uart_print("Error: Packet too small to contain metadata!", VerbosityLevel::ERROR);
        return;
    }

    uart_print("Received " + std::to_string(bytes_read) + " bytes", VerbosityLevel::DEBUG);

    uint8_t received_destination = buffer[0];
    uint8_t received_local_address = buffer[1];

    if (received_destination != lora_address_local) {
        uart_print("Error: Destination address mismatch!", VerbosityLevel::ERROR);
        return;
    }

    if (received_local_address != lora_address_remote) {
        uart_print("Error: Local address mismatch!", VerbosityLevel::ERROR);
        return;
    }

    // Skip 2 bytes being local and remote address appended by ground station
    int start_index = 2;
    std::string received(buffer.begin() + start_index, buffer.end());

    if (received.empty()) return;

    std::stringstream hex_dump;
    hex_dump « "Raw bytes: ";
    for (int i = 0; i < bytes_read; i++) {
        hex_dump « std::hex « std::setfill('0') « std::setw(2)
                « static_cast<int>(buffer[i]) « " ";
    }
    uart_print(hex_dump.str(), VerbosityLevel::DEBUG);

    size_t header_pos = received.find(FRAME_BEGIN);
    size_t footer_pos = received.find(FRAME_END);

    if (header_pos != std::string::npos && footer_pos != std::string::npos && footer_pos > header_pos)
{
        std::string frame_data = received.substr(header_pos, footer_pos + FRAME_END.length() -
header_pos);
        uart_print("Extracted frame (length=" + std::to_string(frame_data.length()) + "): " +
frame_data, VerbosityLevel::DEBUG);
        frame_process(frame_data, Interface::LORA);
    } else {
        uart_print("No valid frame found in received data", VerbosityLevel::WARNING);
    }
}


void handle_uart_input() {
    static std::string uart_buffer;

    while (uart_is_readable(DEBUG_UART_PORT)) {
        char c = uart_getc(DEBUG_UART_PORT);

        if (c == '\r' || c == '\n') {
            uart_print("Received UART string: " + uart_buffer, VerbosityLevel::DEBUG);
            frame_process(uart_buffer, Interface::UART);
```

```
            uart_buffer.clear();
        } else {
            uart_buffer += c;
        }
    }
}
```

8.39 lib/comms/send.cpp File Reference

Implements functions for sending data, including LoRa messages and Frames.

```
#include "communication.h"
```

Functions

- void send_message (string outgoing)

    Sends a message using LoRa.
- void send_frame_lora (const Frame &frame)
- void send_frame_uart (const Frame &frame)

8.39.1 Detailed Description

Implements functions for sending data, including LoRa messages and Frames.

Definition in file send.cpp.

8.39.2 Function Documentation

8.39.2.1 send_message()

```
void send_message (
            string outgoing)
```

Sends a message using LoRa.

Parameters

outgoing	The message to send.

Converts the outgoing string to a C-style string, adds destination and local addresses, and sends the message using LoRa. Prints a log message to the UART.

Definition at line 15 of file send.cpp.

8.39.2.2 send_frame_lora()

```
void send_frame_lora (
            const Frame & frame)
```

Definition at line 39 of file send.cpp.

8.39.2.3 send_frame_uart()

```
void send_frame_uart (
            const Frame & frame)
```

Definition at line 47 of file send.cpp.

8.40 send.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002
00003
00008
00015 void send_message(string outgoing)
00016 {
00017     std::vector<char> send(outgoing.length() + 1);
00018     strcpy(send.data(), outgoing.c_str());
00019
00020     uart_print("LoRa packet begin", VerbosityLevel::DEBUG);
00021     LoRa.beginPacket();         // start packet
00022     LoRa.write(lora_address_remote);  // add destination address
00023     LoRa.write(lora_address_local); // add sender address
00024     LoRa.print(send.data());          // add payload
00025     LoRa.endPacket(false);     // finish packet and send it, param - async
00026
00027     uart_print("LoRa packet end", VerbosityLevel::DEBUG);
00028
00029     std::string message_to_log = "Sent message of size " + std::to_string(send.size());
00030     message_to_log += " to 0x" + std::to_string(lora_address_remote);
00031     message_to_log += " containing: " + string(send.data());
00032
00033     uart_print(message_to_log, VerbosityLevel::DEBUG);
00034
00035     LoRa.flush();
00036 }
00037
00038
00039 void send_frame_lora(const Frame& frame) {
00040     uart_print("Sending frame via LoRa", VerbosityLevel::DEBUG);
00041     string outgoing = frame_encode(frame);
00042     send_message(outgoing);
00043     uart_print("Frame sent via LoRa", VerbosityLevel::DEBUG);
00044 }
00045
00046 // If level is 0 - SILENT it means no diagnostic output but frame communications should still work
00047 void send_frame_uart(const Frame& frame) {
00048     std::string encoded_frame = frame_encode(frame);
00049     uart_print(encoded_frame, VerbosityLevel::SILENT);
00050 }
00051
00052
```

8.41 lib/comms/utils_converters.cpp File Reference

Implements utility functions for converting between different data types.

```
#include "communication.h"
```

Functions

- std::string exception_type_to_string (ExceptionType type)

  Converts an ExceptionType to a string.
- std::string value_unit_type_to_string (ValueUnit unit)

  Converts a ValueUnit to a string.
- std::string operation_type_to_string (OperationType type)

  Converts an OperationType to a string.
- OperationType string_to_operation_type (const std::string &str)

  Converts a string to an OperationType.
- std::string error_code_to_string (ErrorCode code)

  Converts an ErrorCode to its string representation.
- std::vector< uint8_t > hex_string_to_bytes (const std::string &hexString)

  Converts a hex string to a vector of bytes.

8.41.1 Detailed Description

Implements utility functions for converting between different data types.

Definition in file utils_converters.cpp.

8.42 utils_converters.cpp

Go to the documentation of this file.

```
#include "communication.h"



std::string exception_type_to_string(ExceptionType type) {
    switch (type) {
        case ExceptionType::NOT_ALLOWED:       return "NOT ALLOWED";
        case ExceptionType::INVALID_PARAM:     return "INVALID PARAM";
        case ExceptionType::INVALID_OPERATION: return "INVALID OPERATION";
        case ExceptionType::PARAM_UNECESSARY:  return "PARAM UNECESSARY";
        case ExceptionType::NONE:              return "NONE";
        default:                               return "UNKNOWN EXCEPTION";
    }
}


std::string value_unit_type_to_string(ValueUnit unit) {
    switch (unit) {
        case ValueUnit::UNDEFINED:  return "";
        case ValueUnit::SECOND:     return "s";
        case ValueUnit::VOLT:       return "V";
        case ValueUnit::BOOL:       return "";
        case ValueUnit::DATETIME:   return "";
        case ValueUnit::TEXT:       return "";
        case ValueUnit::MILIAMP:    return "mA";
        default: return "";
    }
}


std::string operation_type_to_string(OperationType type) {
    switch (type) {
        case OperationType::GET: return "GET";
        case OperationType::SET: return "SET";
        case OperationType::VAL: return "VAL";
        case OperationType::ERR: return "ERR";
        case OperationType::RES: return "RES";
        case OperationType::SEQ: return "SEQ";
        default: return "UNKNOWN";
    }
}

```

```
OperationType string_to_operation_type(const std::string& str) {
    if (str == "GET") return OperationType::GET;
    if (str == "SET") return OperationType::SET;
    if (str == "VAL") return OperationType::VAL;
    if (str == "ERR") return OperationType::ERR;
    if (str == "RES") return OperationType::RES;
    if (str == "SEQ") return OperationType::SEQ;
    return OperationType::GET; // Default to GET
}

std::string error_code_to_string(ErrorCode code) {
    switch (code) {
        case ErrorCode::PARAM_UNNECESSARY:      return "PARAM_UNNECESSARY";
        case ErrorCode::PARAM_REQUIRED:         return "PARAM_REQUIRED";
        case ErrorCode::PARAM_INVALID:          return "PARAM_INVALID";
        case ErrorCode::INVALID_OPERATION:      return "INVALID_OPERATION";
        case ErrorCode::NOT_ALLOWED:            return "NOT_ALLOWED";
        case ErrorCode::INVALID_FORMAT:         return "INVALID_FORMAT";
        case ErrorCode::INVALID_VALUE:          return "INVALID_VALUE";
        case ErrorCode::FAIL_TO_SET:            return "FAIL_TO_SET";
        case ErrorCode::INTERNAL_FAIL_TO_READ:  return "INTERNAL_FAIL_TO_READ";
        default:                                return "UNKNOWN_ERROR";
    }
}


std::vector<uint8_t> hex_string_to_bytes(const std::string& hexString) {
    std::vector<uint8_t> bytes;
    for (size_t i = 0; i < hexString.length(); i += 2) {
        std::string byteString = hexString.substr(i, 2);
        unsigned int byte;
        std::stringstream ss;
        ss « std::hex « byteString;
        ss » byte;
        bytes.push_back(static_cast<uint8_t>(byte));
    }
    return bytes;
}

```

8.43 lib/eventman/event_manager.cpp File Reference

Implementation of the Event Manager and Event Emitter classes.

```
#include "event_manager.h"
#include <cstdio>
#include "protocol.h"
#include "pico/multicore.h"
#include "communication.h"
#include "utils.h"
#include "DS3231.h"
```

8.43.1 Detailed Description

Implementation of the Event Manager and Event Emitter classes.

This file implements the EventManager class, which provides a singleton instance for logging events to a circular buffer and saving them to persistent storage. The EventEmitter class provides a simple interface for emitting events throughout the system.

Definition in file event_manager.cpp.

8.44 event_manager.cpp

Go to the documentation of this file.

```


#include "event_manager.h"
#include <cstdio>
#include "protocol.h"
#include "pico/multicore.h"
#include "communication.h"
#include "utils.h"
#include "DS3231.h"


bool EventManager::init() {
    if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
        uart_print("Event manager initialized (storage not available)", VerbosityLevel::WARNING);
        return false;
    }

    FILE* file = fopen(EVENT_LOG_FILE, "w");
    if (!file) {
        file = fopen(EVENT_LOG_FILE, "w");
        if (file) {
            fclose(file);
            uart_print("Created new event log", VerbosityLevel::INFO);
        }
        else {
            uart_print("Failed to create event log", VerbosityLevel::ERROR);
            return false;
        }
    }
    else {
        fclose(file);
    }

    uart_print("Event manager initialized", VerbosityLevel::INFO);
    return true;
}


void EventManager::log_event(uint8_t group, uint8_t event) {
    mutex_enter_blocking(&eventMutex);

    uint32_t timestamp = DS3231::get_instance().get_local_time();
    uint16_t id = nextEventId++;

    EventLog& log = events[writeIndex];
    log.id = id;
    log.timestamp = timestamp;
    log.group = group;
    log.event = event;

    writeIndex = (writeIndex + 1) % EVENT_BUFFER_SIZE;
    if (eventCount < EVENT_BUFFER_SIZE) {
        eventCount++;
    }

    eventsSinceFlush++;

    mutex_exit(&eventMutex);

    std::string event_string = "Event: " + std::to_string(id) +
        " Group: " + std::to_string(group) +
        " Event: " + std::to_string(event);
    uart_print(event_string, VerbosityLevel::WARNING);

    if (eventsSinceFlush >= EVENT_FLUSH_THRESHOLD || group == static_cast<uint8_t>(EventGroup::POWER))
 {
        save_to_storage();
        eventsSinceFlush = 0;
    }
}


const EventLog& EventManager::get_event(size_t index) const {
    mutex_enter_blocking(const_cast<mutex_t*>(&eventMutex));
    if (index >= eventCount) {
        static EventLog emptyEvent;
        mutex_exit(const_cast<mutex_t*>(&eventMutex));
        return emptyEvent;
    }

    size_t readIndex;
    if (eventCount == EVENT_BUFFER_SIZE) {
```

```
            readIndex = (writeIndex + index) % EVENT_BUFFER_SIZE;
        }
        else {
            readIndex = index;
        }
        const EventLog& event = events[readIndex];
        mutex_exit(const_cast<mutex_t*>(&eventMutex));
        return event;
}


bool EventManager::save_to_storage() {
        if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
            bool status = fs_init();
            if (!status) {
                return false;
            }
        }

        FILE* file = fopen(EVENT_LOG_FILE, "a");
        if (file) {
            size_t startIdx = (writeIndex >= eventsSinceFlush) ?
                writeIndex - eventsSinceFlush :
                EVENT_BUFFER_SIZE - (eventsSinceFlush - writeIndex);

            for (size_t i = 0; i < eventsSinceFlush; i++) {
                size_t idx = (startIdx + i) % EVENT_BUFFER_SIZE;
                fprintf(file, "%u;%lu;%u;%u\n",
                    events[idx].id,
                    events[idx].timestamp,
                    events[idx].group,
                    events[idx].event
                );
            }
            fclose(file);
            uart_print("Events saved to storage", VerbosityLevel::INFO);
            return true;
        }
        return false;
}

```

8.45 lib/eventman/event_manager.h File Reference

Header file for the Event Manager and Event Emitter classes.

```
#include "PowerManager.h"
#include <cstdint>
#include <string>
#include "pico/mutex.h"
#include "storage.h"
#include "utils.h"
#include "system_state_manager.h"
```

Classes

- class EventLog

  Structure for storing event log data.
- class EventManager

  Manages event logging and storage.
- class EventEmitter

  Provides a simple interface for emitting events.

Macros

- #define EVENT_BUFFER_SIZE 100

  Size of the event buffer.
- #define EVENT_FLUSH_THRESHOLD 10

  Number of events to accumulate before flushing to storage.
- #define EVENT_LOG_FILE "/event_log.csv"

  Path to the event log file.

Enumerations

- enum class EventGroup : uint8_t {
  EventGroup::SYSTEM = 0x00 , EventGroup::POWER = 0x01 , EventGroup::COMMS = 0x02 ,
  EventGroup::GPS = 0x03 ,
  EventGroup::CLOCK = 0x04 }

  Enumeration of event groups.
- enum class SystemEvent : uint8_t {
  SystemEvent::BOOT = 0x01 , SystemEvent::SHUTDOWN = 0x02 , SystemEvent::WATCHDOG_RESET =
  0x03 , SystemEvent::CORE1_START = 0x04 ,
  SystemEvent::CORE1_STOP = 0x05 }

  Enumeration of system events.
- enum class PowerEvent : uint8_t {
  PowerEvent::BATTERY_LOW = 0x01 , PowerEvent::BATTERY_FULL = 0x02 , PowerEvent::POWER_FALLING
  = 0x03 , PowerEvent::BATTERY_NORMAL = 0x04 ,
  PowerEvent::SOLAR_ACTIVE = 0x05 , PowerEvent::SOLAR_INACTIVE = 0x06 , PowerEvent::USB_CONNECTED
  = 0x07 , PowerEvent::USB_DISCONNECTED = 0x08 }

  Enumeration of power events.
- enum class CommsEvent : uint8_t {
  CommsEvent::RADIO_INIT = 0x01 , CommsEvent::RADIO_ERROR = 0x02 , CommsEvent::MSG_RECEIVED
  = 0x03 , CommsEvent::MSG_SENT = 0x04 ,
  CommsEvent::UART_ERROR = 0x06 }

  Enumeration of communications events.
- enum class GPSEvent : uint8_t {
  GPSEvent::LOCK = 0x01 , GPSEvent::LOST = 0x02 , GPSEvent::ERROR = 0x03 , GPSEvent::POWER_ON
  = 0x04 ,
  GPSEvent::POWER_OFF = 0x05 , GPSEvent::DATA_READY = 0x06 , GPSEvent::PASS_THROUGH_START
  = 0x07 , GPSEvent::PASS_THROUGH_END = 0x08 }

  Enumeration of GPS events.
- enum class ClockEvent : uint8_t { ClockEvent::CHANGED = 0x01 , ClockEvent::GPS_SYNC = 0x02 ,
  ClockEvent::GPS_SYNC_DATA_NOT_READY = 0x03 }

  Enumeration of clock events.

Functions

- class EventLog __attribute__ ((packed))

Variables

- uint16_t id

  Unique identifier for the event.
- uint32_t timestamp

  Timestamp of the event in milliseconds since boot.
- uint8_t group

  Event group.
- uint8_t event

  Event code.
- class EventManager __attribute__

8.45.1 Detailed Description

Header file for the Event Manager and Event Emitter classes.

This file defines the classes and enumerations necessary for managing and emitting system events. The EventManager class provides a singleton instance for logging events to a circular buffer and saving them to persistent storage. The EventEmitter class provides a simple interface for emitting events throughout the system.

Definition in file event_manager.h.

8.45.2 Variable Documentation

8.45.2.1 id

```
uint16_t id
```

Unique identifier for the event.

Definition at line 2 of file event_manager.h.

8.45.2.2 timestamp

```
uint32_t timestamp
```

Timestamp of the event in milliseconds since boot.

Definition at line 4 of file event_manager.h.

8.45.2.3 group

```
uint8_t group
```

Event group.

Definition at line 6 of file event_manager.h.

8.45.2.4 event

```
uint8_t event
```

Event code.

Definition at line 8 of file event_manager.h.

8.46 event_manager.h

Go to the documentation of this file.

```
#ifndef EVENT_MANAGER_H
#define EVENT_MANAGER_H

#include "PowerManager.h"
#include <cstdint>
#include <string>
#include "pico/mutex.h"
#include "storage.h"
#include "utils.h"
#include "system_state_manager.h"

#define EVENT_BUFFER_SIZE 100

#define EVENT_FLUSH_THRESHOLD 10

#define EVENT_LOG_FILE "/event_log.csv"


enum class EventGroup : uint8_t {
    SYSTEM = 0x00,
    POWER = 0x01,
    COMMS = 0x02,
    GPS = 0x03,
    CLOCK = 0x04
};


enum class SystemEvent : uint8_t {
    BOOT = 0x01,
    SHUTDOWN = 0x02,
    WATCHDOG_RESET = 0x03,
    CORE1_START = 0x04,
    CORE1_STOP = 0x05
};

enum class PowerEvent : uint8_t {
    BATTERY_LOW = 0x01,
    BATTERY_FULL = 0x02,
    POWER_FALLING = 0x03,
    BATTERY_NORMAL = 0x04,
    SOLAR_ACTIVE = 0x05,
    SOLAR_INACTIVE = 0x06,
    USB_CONNECTED = 0x07,
    USB_DISCONNECTED = 0x08
};


enum class CommsEvent : uint8_t {
    RADIO_INIT = 0x01,
    RADIO_ERROR = 0x02,
    MSG_RECEIVED = 0x03,
    MSG_SENT = 0x04,
    UART_ERROR = 0x06
};

enum class GPSEvent : uint8_t {
    LOCK = 0x01,
    LOST = 0x02,
    ERROR = 0x03,
    POWER_ON = 0x04,
    POWER_OFF = 0x05,
    DATA_READY = 0x06,
    PASS_THROUGH_START = 0x07,
    PASS_THROUGH_END = 0x08
};

enum class ClockEvent : uint8_t {
    CHANGED = 0x01,
    GPS_SYNC = 0x02,
    GPS_SYNC_DATA_NOT_READY = 0x03
};


class EventLog {
    public:
        uint16_t id;
        uint32_t timestamp;
        uint8_t group;
        uint8_t event;
    } __attribute__((packed));
```

```cpp


class EventManager {
private:
    EventLog events[EVENT_BUFFER_SIZE];
    size_t eventCount;
    size_t writeIndex;
    mutex_t eventMutex;
    uint16_t nextEventId;
    size_t eventsSinceFlush;

    EventManager() :
        eventCount(0),
        writeIndex(0),
        nextEventId(0),
        eventsSinceFlush(0)
    {
        mutex_init(&eventMutex);
    }

    EventManager(const EventManager&) = delete;
    EventManager& operator=(const EventManager&) = delete;

    public:
    static EventManager& get_instance() {
        static EventManager instance;
        return instance;
    }

    bool init();

    void log_event(uint8_t group, uint8_t event);

    const EventLog& get_event(size_t index) const;

    size_t get_event_count() const { return eventCount; }

    bool save_to_storage();

    bool load_from_storage();
};

class EventEmitter {
public:
    template<typename T>
    static void emit(EventGroup group, T event) {
        EventManager::get_instance().log_event(
            static_cast<uint8_t>(group),
            static_cast<uint8_t>(event)
        );
    }
};

#endif // EVENT_MANAGER_H
```

8.47 lib/location/gps_collector.cpp File Reference

Implementation of the GPS data collector module.

```cpp
#include "lib/location/gps_collector.h"
#include "utils.h"
#include "pico/time.h"
#include "lib/location/NMEA/nmea_data.h"
#include "event_manager.h"
#include <vector>
#include <ctime>
#include <cstring>
#include "DS3231.h"
#include <sstream>
#include "system_state_manager.h"
```

Macros

- #define MAX_RAW_DATA_LENGTH 256

  Maximum length of the raw data buffer for NMEA sentences.

Functions

- std::vector< std::string > splitString (const std::string &str, char delimiter)

  Splits a string into tokens based on a delimiter.
- void collect_gps_data ()

  Collects GPS data from the UART and updates the NMEA data.

8.47.1 Detailed Description

Implementation of the GPS data collector module.

This file implements the function `collect_gps_data`, which is responsible for reading raw NMEA sentences from the GPS UART, parsing them, and updating the NMEA data in the NMEAData singleton.

Definition in file gps_collector.cpp.

8.48 gps_collector.cpp

Go to the documentation of this file.

```
#include "lib/location/gps_collector.h"
#include "utils.h"
#include "pico/time.h"
#include "lib/location/NMEA/nmea_data.h"
#include "event_manager.h"
#include <vector>
#include <ctime>
#include <cstring>
#include "DS3231.h"
#include <sstream>
#include "system_state_manager.h"

#define MAX_RAW_DATA_LENGTH 256

std::vector<std::string> splitString(const std::string& str, char delimiter) {
    std::vector<std::string> tokens;
    std::stringstream ss(str);
    std::string token;
    while (std::getline(ss, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}

void collect_gps_data() {

    if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
        return;
    }

    static char raw_data_buffer[MAX_RAW_DATA_LENGTH];
    static int raw_data_index = 0;

    while (uart_is_readable(GPS_UART_PORT)) {
        char c = uart_getc(GPS_UART_PORT);

        if (c == '\r' || c == '\n') {
            // End of message
            if (raw_data_index > 0) {
                raw_data_buffer[raw_data_index] = '\0';
```

```
                std::string message(raw_data_buffer);
                raw_data_index = 0;

                // Split the message into tokens
                std::vector<std::string> tokens = splitString(message, ',');

                // Update the global vectors based on the sentence type
                if (message.find("$GPRMC") == 0) {
                    NMEAData::get_instance().update_rmc_tokens(tokens);
                } else if (message.find("$GPGGA") == 0) {
                    NMEAData::get_instance().update_gga_tokens(tokens);
                }
            }
        } else {
            // Append to buffer
            if (raw_data_index < MAX_RAW_DATA_LENGTH - 1) {
                raw_data_buffer[raw_data_index++] = c;
            } else {
                raw_data_index = 0;
            }
        }
    }
}
```

8.49 lib/location/gps_collector.h File Reference

Header file for the GPS data collector module.

```
#include <string>
#include "hardware/uart.h"
#include "lib/location/NMEA/nmea_data.h"
#include "pin_config.h"
```

Functions

- void collect_gps_data ()

  Collects GPS data from the UART and updates the NMEA data.

8.49.1 Detailed Description

Header file for the GPS data collector module.

This file defines the function `collect_gps_data`, which is responsible for reading raw NMEA sentences from the GPS UART, parsing them, and updating the NMEA data in the NMEAData singleton.

Definition in file gps_collector.h.

8.50 gps_collector.h

Go to the documentation of this file.

```

#ifndef GPS_COLLECTOR_H
#define GPS_COLLECTOR_H

#include <string>
#include "hardware/uart.h"
#include "lib/location/NMEA/nmea_data.h" // Include the new header
#include "pin_config.h"

void collect_gps_data();

#endif
```

8.51 lib/location/NMEA/NMEA_data.h File Reference

Header file for the NMEAData class, which manages parsed NMEA sentences.

```
#include <vector>
#include <string>
#include "pico/sync.h"
#include <ctime>
#include <cstring>
```

Classes

- class NMEAData

  Manages parsed NMEA sentences.

8.51.1 Detailed Description

Header file for the NMEAData class, which manages parsed NMEA sentences.

This file defines the NMEAData class, a singleton that stores and provides access to parsed data from NMEA sentences received from a GPS module. It includes methods for updating and retrieving RMC and GGA tokens, as well as converting the data to a Unix timestamp.

Definition in file NMEA_data.h.

8.52 NMEA_data.h

Go to the documentation of this file.

```
#ifndef NMEA_DATA_H
#define NMEA_DATA_H
#include <vector>
#include <string>
#include "pico/sync.h"
#include <ctime>
#include <cstring>
class NMEAData {
private:
    std::vector<std::string> rmc_tokens_;
    std::vector<std::string> gga_tokens_;
    mutex_t rmc_mutex_;
    mutex_t gga_mutex_;
    NMEAData() {
        mutex_init(&rmc_mutex_);
        mutex_init(&gga_mutex_);
    }
    NMEAData(const NMEAData&) = delete;
    NMEAData& operator=(const NMEAData&) = delete;
public:
    static NMEAData& get_instance() {
        static NMEAData instance;
        return instance;
    }
    void update_rmc_tokens(const std::vector<std::string>& tokens) {
        mutex_enter_blocking(&rmc_mutex_);
```

```
        rmc_tokens_ = tokens;
        mutex_exit(&rmc_mutex_);
    }

    void update_gga_tokens(const std::vector<std::string>& tokens) {
        mutex_enter_blocking(&gga_mutex_);
        gga_tokens_ = tokens;
        mutex_exit(&gga_mutex_);
    }

    std::vector<std::string> get_rmc_tokens() const {
        mutex_enter_blocking(const_cast<mutex_t*>(&rmc_mutex_));
        std::vector<std::string> copy = rmc_tokens_;
        mutex_exit(const_cast<mutex_t*>(&rmc_mutex_));
        return copy;
    }

    std::vector<std::string> get_gga_tokens() const {
        mutex_enter_blocking(const_cast<mutex_t*>(&gga_mutex_));
        std::vector<std::string> copy = gga_tokens_;
        mutex_exit(const_cast<mutex_t*>(&gga_mutex_));
        return copy;
    }

    bool has_valid_time() const {
        return rmc_tokens_.size() >= 10 && rmc_tokens_[1].length() > 5;
    }

    time_t get_unix_time() const {
        if (!has_valid_time()) {
            return 0; // Invalid time
        }

        // Parse date and time from RMC tokens
        // Format: hhmmss.sss,A,ddmm.mmmm,N,dddmm.mmmm,W,speed,course,ddmmyy
        std::string time_str = rmc_tokens_[1]; // hhmmss.sss
        std::string date_str = rmc_tokens_[9]; // ddmmyy

        if (time_str.length() < 6 || date_str.length() < 6) {
            return 0;
        }

        struct tm timeinfo;
        memset(&timeinfo, 0, sizeof(timeinfo));

        // Parse time: hours (0-1), minutes (2-3), seconds (4-5)
        timeinfo.tm_hour = std::stoi(time_str.substr(0, 2));
        timeinfo.tm_min = std::stoi(time_str.substr(2, 2));
        timeinfo.tm_sec = std::stoi(time_str.substr(4, 2));

        // Parse date: day (0-1), month (2-3), year (4-5)
        timeinfo.tm_mday = std::stoi(date_str.substr(0, 2));
        timeinfo.tm_mon = std::stoi(date_str.substr(2, 2)) - 1; // Month is 0-11
        timeinfo.tm_year = std::stoi(date_str.substr(4, 2)) + 100; // Year is since 1900

        return mktime(&timeinfo);
    }
};

#endif
```

8.53 lib/pin_config.h File Reference

```
#include <stdint.h>
```

Macros

- #define DEBUG_UART_PORT uart0
- #define DEBUG_UART_BAUD_RATE 115200
- #define DEBUG_UART_TX_PIN 0
- #define DEBUG_UART_RX_PIN 1
- #define MAIN_I2C_PORT i2c1
- #define MAIN_I2C_SDA_PIN 6

- #define MAIN_I2C_SCL_PIN 7
- #define GPS_UART_PORT uart1
- #define GPS_UART_BAUD_RATE 9600
- #define GPS_UART_TX_PIN 8
- #define GPS_UART_RX_PIN 9
- #define GPS_POWER_ENABLE_PIN 14
- #define SENSORS_POWER_ENABLE_PIN 15
- #define SENSORS_I2C_PORT i2c0
- #define SENSORS_I2C_SDA_PIN 4
- #define SENSORS_I2C_SCL_PIN 5
- #define BUFFER_SIZE 85
- #define SD_SPI_PORT spi1
- #define SD_MISO_PIN 12
- #define SD_MOSI_PIN 11
- #define SD_SCK_PIN 10
- #define SD_CS_PIN 13
- #define SD_CARD_DETECT_PIN 28
- #define SX1278_MISO 16
- #define SX1278_CS 17
- #define SX1278_SCK 18
- #define SX1278_MOSI 19
- #define SPI_PORT spi0
- #define READ_BIT 0x80
- #define LORA_DEFAULT_SPI spi0
- #define LORA_DEFAULT_SPI_FREQUENCY 8E6
- #define LORA_DEFAULT_SS_PIN 17
- #define LORA_DEFAULT_RESET_PIN 22
- #define LORA_DEFAULT_DIO0_PIN 20
- #define PA_OUTPUT_RFO_PIN 11
- #define PA_OUTPUT_PA_BOOST_PIN 12

Variables

- constexpr int lora_cs_pin = 17
- constexpr int lora_reset_pin = 22
- constexpr int lora_irq_pin = 28
- uint8_t lora_address_local = 37
- uint8_t lora_address_remote = 21

8.53.1 Macro Definition Documentation

8.53.1.1 DEBUG_UART_PORT

```
#define DEBUG_UART_PORT uart0
```

Definition at line 8 of file pin_config.h.

8.53.1.2 DEBUG_UART_BAUD_RATE

```
#define DEBUG_UART_BAUD_RATE 115200
```

Definition at line 9 of file pin_config.h.

8.53.1.3 DEBUG_UART_TX_PIN

```
#define DEBUG_UART_TX_PIN 0
```

Definition at line 11 of file pin_config.h.

8.53.1.4 DEBUG_UART_RX_PIN

```
#define DEBUG_UART_RX_PIN 1
```

Definition at line 12 of file pin_config.h.

8.53.1.5 MAIN_I2C_PORT

```
#define MAIN_I2C_PORT i2c1
```

Definition at line 14 of file pin_config.h.

8.53.1.6 MAIN_I2C_SDA_PIN

```
#define MAIN_I2C_SDA_PIN 6
```

Definition at line 15 of file pin_config.h.

8.53.1.7 MAIN_I2C_SCL_PIN

```
#define MAIN_I2C_SCL_PIN 7
```

Definition at line 16 of file pin_config.h.

8.53.1.8 GPS_UART_PORT

```
#define GPS_UART_PORT uart1
```

Definition at line 19 of file pin_config.h.

8.53.1.9 GPS_UART_BAUD_RATE

```
#define GPS_UART_BAUD_RATE 9600
```

Definition at line 20 of file pin_config.h.

8.53.1.10 GPS_UART_TX_PIN

```
#define GPS_UART_TX_PIN 8
```

Definition at line 21 of file pin_config.h.

8.53.1.11 GPS_UART_RX_PIN

```
#define GPS_UART_RX_PIN 9
```

Definition at line 22 of file pin_config.h.

8.53.1.12 GPS_POWER_ENABLE_PIN

```
#define GPS_POWER_ENABLE_PIN 14
```

Definition at line 23 of file pin_config.h.

8.53.1.13 SENSORS_POWER_ENABLE_PIN

```
#define SENSORS_POWER_ENABLE_PIN 15
```

Definition at line 25 of file pin_config.h.

8.53.1.14 SENSORS_I2C_PORT

```
#define SENSORS_I2C_PORT i2c0
```

Definition at line 26 of file pin_config.h.

8.53.1.15 SENSORS_I2C_SDA_PIN

```
#define SENSORS_I2C_SDA_PIN 4
```

Definition at line 27 of file pin_config.h.

8.53.1.16 SENSORS_I2C_SCL_PIN

```
#define SENSORS_I2C_SCL_PIN 5
```

Definition at line 28 of file pin_config.h.

8.53.1.17 BUFFER_SIZE

```
#define BUFFER_SIZE 85
```

Definition at line 30 of file pin_config.h.

8.53.1.18 SD_SPI_PORT

```
#define SD_SPI_PORT spi1
```

Definition at line 33 of file pin_config.h.

8.53.1.19 SD_MISO_PIN

```
#define SD_MISO_PIN 12
```

Definition at line 34 of file pin_config.h.

8.53.1.20 SD_MOSI_PIN

```
#define SD_MOSI_PIN 11
```

Definition at line 35 of file pin_config.h.

8.53.1.21 SD_SCK_PIN

```
#define SD_SCK_PIN 10
```

Definition at line 36 of file pin_config.h.

8.53.1.22 SD_CS_PIN

```
#define SD_CS_PIN 13
```

Definition at line 37 of file pin_config.h.

8.53.1.23 SD_CARD_DETECT_PIN

```
#define SD_CARD_DETECT_PIN 28
```

Definition at line 38 of file pin_config.h.

8.53.1.24 SX1278_MISO

```
#define SX1278_MISO 16
```

Definition at line 40 of file pin_config.h.

8.53.1.25 SX1278_CS

```
#define SX1278_CS 17
```

Definition at line 41 of file pin_config.h.

8.53.1.26 SX1278_SCK

```
#define SX1278_SCK 18
```

Definition at line 42 of file pin_config.h.

8.53.1.27 SX1278_MOSI

```
#define SX1278_MOSI 19
```

Definition at line 43 of file pin_config.h.

8.53.1.28 SPI_PORT

```
#define SPI_PORT spi0
```

Definition at line 45 of file pin_config.h.

8.53.1.29 READ_BIT

```
#define READ_BIT 0x80
```

Definition at line 46 of file pin_config.h.

8.53.1.30 LORA_DEFAULT_SPI

```
#define LORA_DEFAULT_SPI spi0
```

Definition at line 48 of file pin_config.h.

8.53.1.31 LORA_DEFAULT_SPI_FREQUENCY

```
#define LORA_DEFAULT_SPI_FREQUENCY 8E6
```

Definition at line 49 of file pin_config.h.

8.53.1.32 LORA_DEFAULT_SS_PIN

```
#define LORA_DEFAULT_SS_PIN 17
```

Definition at line 50 of file pin_config.h.

8.53.1.33 LORA_DEFAULT_RESET_PIN

```
#define LORA_DEFAULT_RESET_PIN 22
```

Definition at line 51 of file pin_config.h.

8.53.1.34 LORA_DEFAULT_DIO0_PIN

```
#define LORA_DEFAULT_DIO0_PIN 20
```

Definition at line 52 of file pin_config.h.

8.53.1.35 PA_OUTPUT_RFO_PIN

```
#define PA_OUTPUT_RFO_PIN 11
```

Definition at line 54 of file pin_config.h.

8.53.1.36 PA_OUTPUT_PA_BOOST_PIN

```
#define PA_OUTPUT_PA_BOOST_PIN 12
```

Definition at line 55 of file pin_config.h.

8.53.2 Variable Documentation

8.53.2.1 lora_cs_pin

```
int lora_cs_pin = 17 [inline], [constexpr]
```

Definition at line 57 of file pin_config.h.

8.53.2.2 lora_reset_pin

```
int lora_reset_pin = 22 [inline], [constexpr]
```

Definition at line 58 of file pin_config.h.

8.53.2.3 lora_irq_pin

```
int lora_irq_pin = 28 [inline], [constexpr]
```

Definition at line 59 of file pin_config.h.

8.53.2.4 lora_address_local

```
uint8_t lora_address_local = 37 [inline]
```

Definition at line 61 of file pin_config.h.

8.53.2.5 lora_address_remote

```
uint8_t lora_address_remote = 21 [inline]
```

Definition at line 62 of file pin_config.h.

8.54 pin_config.h

Go to the documentation of this file.

```

#ifndef PIN_CONFIG_H
#define PIN_CONFIG_H

#include <stdint.h>

//DEBUG uart
#define DEBUG_UART_PORT uart0
#define DEBUG_UART_BAUD_RATE 115200

#define DEBUG_UART_TX_PIN 0
#define DEBUG_UART_RX_PIN 1

#define MAIN_I2C_PORT i2c1
#define MAIN_I2C_SDA_PIN 6
#define MAIN_I2C_SCL_PIN 7

// GPS configuration
#define GPS_UART_PORT uart1
#define GPS_UART_BAUD_RATE 9600
#define GPS_UART_TX_PIN 8
#define GPS_UART_RX_PIN 9
#define GPS_POWER_ENABLE_PIN 14

#define SENSORS_POWER_ENABLE_PIN 15
#define SENSORS_I2C_PORT i2c0
#define SENSORS_I2C_SDA_PIN 4
#define SENSORS_I2C_SCL_PIN 5

#define BUFFER_SIZE 85

// SPI configuration for SD card
#define SD_SPI_PORT spi1
#define SD_MISO_PIN 12
#define SD_MOSI_PIN 11
#define SD_SCK_PIN 10
#define SD_CS_PIN 13
#define SD_CARD_DETECT_PIN 28

#define SX1278_MISO 16
#define SX1278_CS   17
#define SX1278_SCK  18
#define SX1278_MOSI 19

#define SPI_PORT spi0
#define READ_BIT 0x80

#define LORA_DEFAULT_SPI           spi0
#define LORA_DEFAULT_SPI_FREQUENCY 8E6
#define LORA_DEFAULT_SS_PIN        17
#define LORA_DEFAULT_RESET_PIN     22
#define LORA_DEFAULT_DIO0_PIN      20

#define PA_OUTPUT_RFO_PIN          11
#define PA_OUTPUT_PA_BOOST_PIN     12

inline constexpr int lora_cs_pin = 17;          // LoRa radio chip select
inline constexpr int lora_reset_pin = 22;       // LoRa radio reset
inline constexpr int lora_irq_pin = 28;         // LoRa hardware interrupt pin

inline uint8_t lora_address_local = 37;         // address of this device
inline uint8_t lora_address_remote = 21;        // destination to send to

#endif
```

8.55 lib/powerman/INA3221/INA3221.cpp File Reference

Implementation of the INA3221 power monitor driver.

```
#include "INA3221.h"
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
```

```
#include <iostream>
#include "pin_config.h"
#include "utils.h"
#include <sstream>
```

8.55.1 Detailed Description

Implementation of the INA3221 power monitor driver.

This file contains the implementation for the INA3221 triple-channel power monitor, providing functionality for voltage, current, and power monitoring with alert capabilities.

Definition in file INA3221.cpp.

8.56 INA3221.cpp

Go to the documentation of this file.

```
#include "INA3221.h"
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include <iostream>
#include "pin_config.h"
#include "utils.h"
#include <sstream>






INA3221::INA3221(ina3221_addr_t addr, i2c_inst_t* i2c)
    : _i2c_addr(addr), _i2c(i2c) {}


bool INA3221::begin() {
    uart_print("INA3221 initializing...", VerbosityLevel::DEBUG);

    _shuntRes[0] = 10;
    _shuntRes[1] = 10;
    _shuntRes[2] = 10;

    _filterRes[0] = 10;
    _filterRes[1] = 10;
    _filterRes[2] = 10;

    uint16_t manuf_id = get_manufacturer_id();
    uint16_t die_id = get_die_id();
    std::stringstream ss;
    ss « "INA3221 Manufacturer ID: 0x" « std::hex « manuf_id
              « ", Die ID: 0x" « die_id;
    uart_print(ss.str(), VerbosityLevel::INFO);

    if (manuf_id == 0x5449 && die_id == 0x3220) {
       uart_print("INA3221 found and initialized.", VerbosityLevel::DEBUG);
        return true;
    } else {
        uart_print("INA3221 initialization failed. Incorrect IDs.", VerbosityLevel::ERROR);
        return false;
    }
}


void INA3221::reset(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.reset = 1;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}
```

```


uint16_t INA3221::get_manufacturer_id() {
    uint16_t id = 0;
    _read(INA3221_REG_MANUF_ID, &id);
    return id;
}


uint16_t INA3221::get_die_id() {
    uint16_t id = 0;
    _read(INA3221_REG_DIE_ID, &id);
    return id;
}


uint16_t INA3221::read_register(ina3221_reg_t reg){
    uint16_t val = 0;
    _read(reg, &val);
    return val;
}


//configure

void INA3221::set_mode_power_down(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_bus_en = 0;
    conf_reg.mode_continious_en =0 ;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_mode_continuous(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_continious_en =1;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_mode_triggered(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_continious_en = 0;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_shunt_measurement_enable(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_shunt_en = 1;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_shunt_measurement_disable(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_shunt_en = 0;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_bus_measurement_enable(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_bus_en = 1;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_bus_measurement_disable(){
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.mode_bus_en = 0;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}
```

```
void INA3221::set_averaging_mode(ina3221_avg_mode_t mode) {
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.avg_mode = mode;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_bus_conversion_time(ina3221_conv_time_t convTime) {
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.bus_conv_time = convTime;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


void INA3221::set_shunt_conversion_time(ina3221_conv_time_t convTime) {
    conf_reg_t conf_reg;

    _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
    conf_reg.shunt_conv_time = convTime;
    _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
}


//get measurement
int32_t INA3221::get_shunt_voltage(ina3221_ch_t channel) {
    int32_t res;
    ina3221_reg_t reg;
    uint16_t val_raw = 0;

    switch(channel){
        case INA3221_CH1:
            reg = INA3221_REG_CH1_SHUNTV;
            break;
        case INA3221_CH2:
            reg = INA3221_REG_CH2_SHUNTV;
            break;
        case INA3221_CH3:
            reg = INA3221_REG_CH3_SHUNTV;
            break;
    }

    _read(reg, &val_raw);

    res = (int16_t) (val_raw » 3);
    res *= SHUNT_VOLTAGE_LSB_UV;

    return res;
}


float INA3221::get_current_ma(ina3221_ch_t channel) {
    int32_t shunt_uV = 0;
    float current_A = 0;

    shunt_uV = get_shunt_voltage(channel);
    current_A = shunt_uV / (int32_t)_shuntRes[channel];
    return current_A;
}


float INA3221::get_voltage(ina3221_ch_t channel) {
    float voltage_V = 0.0;
    ina3221_reg_t reg;
    uint16_t val_raw = 0;

    switch(channel){
        case INA3221_CH1:
            reg = INA3221_REG_CH1_BUSV;
            break;
        case INA3221_CH2:
            reg = INA3221_REG_CH2_BUSV;
            break;
        case INA3221_CH3:
            reg = INA3221_REG_CH3_BUSV;
            break;
    }

    _read(reg, &val_raw);
    voltage_V = val_raw / 1000.0;
    return voltage_V;
}
```

```
// private
void INA3221::_read(ina3221_reg_t reg, uint16_t *val) {
    uint8_t reg_buf = reg;
    uint8_t data[2];

    int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, &reg_buf, 1, true);
    if (ret != 1) {
        std::cerr « "Failed to write register address to I2C device." « std::endl;
        return;
    }

    ret = i2c_read_blocking(MAIN_I2C_PORT, _i2c_addr, data, 2, false);
    if (ret != 2) {
        std::cerr « "Failed to read data from I2C device." « std::endl;
        return;
    }

    *val = (data[0] « 8) | data[1];
}


void INA3221::_write(ina3221_reg_t reg, uint16_t *val) {
    uint8_t buf[3];
    buf[0] = reg;
    buf[1] = (*val » 8) & 0xFF; // MSB
    buf[2] = (*val) & 0xFF;      // LSB

    int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, buf, 3, false);
    if (ret != 3) {
        std::cerr « "Failed to write data to I2C device." « std::endl;
    }
}
```

8.57 lib/powerman/INA3221/INA3221.h File Reference

Header file for the INA3221 triple-channel power monitor driver.

```
#include <stdint.h>
#include <iostream>
#include <hardware/i2c.h>
```

Classes

- class INA3221

    INA3221 Triple-Channel Power Monitor driver class.
- struct INA3221::conf_reg_t

    Configuration register bit fields.
- struct INA3221::masken_reg_t

    Mask/Enable register bit fields.

Enumerations

- enum ina3221_addr_t { INA3221_ADDR40_GND = 0b1000000 , INA3221_ADDR41_VCC = 0b1000001 , INA3221_ADDR42_SDA = 0b1000010 , INA3221_ADDR43_SCL = 0b1000011 }
- enum ina3221_ch_t { INA3221_CH1 = 0 , INA3221_CH2 , INA3221_CH3 }

- enum ina3221_reg_t {
INA3221_REG_CONF = 0 , INA3221_REG_CH1_SHUNTV , INA3221_REG_CH1_BUSV , INA3221_REG_CH2_SHUNTV
,
INA3221_REG_CH2_BUSV , INA3221_REG_CH3_SHUNTV , INA3221_REG_CH3_BUSV , INA3221_REG_CH1_CRIT_ALER
,
INA3221_REG_CH1_WARNING_ALERT_LIM , INA3221_REG_CH2_CRIT_ALERT_LIM , INA3221_REG_CH2_WARNING_A
, INA3221_REG_CH3_CRIT_ALERT_LIM ,
INA3221_REG_CH3_WARNING_ALERT_LIM , INA3221_REG_SHUNTV_SUM , INA3221_REG_SHUNTV_SUM_LIM
, INA3221_REG_MASK_ENABLE ,
INA3221_REG_PWR_VALID_HI_LIM , INA3221_REG_PWR_VALID_LO_LIM , INA3221_REG_MANUF_ID
= 0xFE , INA3221_REG_DIE_ID = 0xFF }

  Register addresses for INA3221.
- enum ina3221_conv_time_t {
INA3221_REG_CONF_CT_140US = 0 , INA3221_REG_CONF_CT_204US , INA3221_REG_CONF_CT_332US
, INA3221_REG_CONF_CT_588US ,
INA3221_REG_CONF_CT_1100US , INA3221_REG_CONF_CT_2116US , INA3221_REG_CONF_CT_4156US
, INA3221_REG_CONF_CT_8244US }

  Conversion time settings.
- enum ina3221_avg_mode_t {
INA3221_REG_CONF_AVG_1 = 0 , INA3221_REG_CONF_AVG_4 , INA3221_REG_CONF_AVG_16 ,
INA3221_REG_CONF_AVG_64 ,
INA3221_REG_CONF_AVG_128 , INA3221_REG_CONF_AVG_256 , INA3221_REG_CONF_AVG_512 ,
INA3221_REG_CONF_AVG_1024 }

  Averaging mode settings.

Variables

- const int INA3221_CH_NUM = 3

  Number of channels in INA3221.
- const int SHUNT_VOLTAGE_LSB_UV = 5

  LSB value for shunt voltage measurements in microvolts.

8.57.1 Detailed Description

Header file for the INA3221 triple-channel power monitor driver.

Definition in file INA3221.h.

8.57.2 Enumeration Type Documentation

8.57.2.1 ina3221_addr_t

```
enum ina3221_addr_t
```

Enumerator

INA3221_ADDR40_GND	
INA3221_ADDR41_VCC	
INA3221_ADDR42_SDA	
INA3221_ADDR43_SCL	

Definition at line 12 of file INA3221.h.

8.57.2.2 ina3221_ch_t

```
enum ina3221_ch_t
```

Enumerator

INA3221_CH1	
INA3221_CH2	
INA3221_CH3	

Definition at line 23 of file INA3221.h.

8.57.2.3 ina3221_reg_t

```
enum ina3221_reg_t
```

Register addresses for INA3221.

Enumerator

INA3221_REG_CONF	
INA3221_REG_CH1_SHUNTV	
INA3221_REG_CH1_BUSV	
INA3221_REG_CH2_SHUNTV	
INA3221_REG_CH2_BUSV	
INA3221_REG_CH3_SHUNTV	
INA3221_REG_CH3_BUSV	
INA3221_REG_CH1_CRIT_ALERT_LIM	
INA3221_REG_CH1_WARNING_ALERT_LIM	
INA3221_REG_CH2_CRIT_ALERT_LIM	
INA3221_REG_CH2_WARNING_ALERT_LIM	
INA3221_REG_CH3_CRIT_ALERT_LIM	
INA3221_REG_CH3_WARNING_ALERT_LIM	
INA3221_REG_SHUNTV_SUM	
INA3221_REG_SHUNTV_SUM_LIM	
INA3221_REG_MASK_ENABLE	
INA3221_REG_PWR_VALID_HI_LIM	
INA3221_REG_PWR_VALID_LO_LIM	
INA3221_REG_MANUF_ID	
INA3221_REG_DIE_ID	

Definition at line 38 of file INA3221.h.

8.57.2.4 ina3221_conv_time_t

```
enum ina3221_conv_time_t
```

Conversion time settings.

Time taken for each measurement conversion

Enumerator

INA3221_REG_CONF_CT_140US	
INA3221_REG_CONF_CT_204US	
INA3221_REG_CONF_CT_332US	
INA3221_REG_CONF_CT_588US	
INA3221_REG_CONF_CT_1100US	
INA3221_REG_CONF_CT_2116US	
INA3221_REG_CONF_CT_4156US	
INA3221_REG_CONF_CT_8244US	

Definition at line 65 of file INA3221.h.

8.57.2.5 ina3221_avg_mode_t

```
enum ina3221_avg_mode_t
```

Averaging mode settings.

Number of samples to average for each measurement

Enumerator

INA3221_REG_CONF_AVG_1	
INA3221_REG_CONF_AVG_4	
INA3221_REG_CONF_AVG_16	
INA3221_REG_CONF_AVG_64	
INA3221_REG_CONF_AVG_128	
INA3221_REG_CONF_AVG_256	
INA3221_REG_CONF_AVG_512	
INA3221_REG_CONF_AVG_1024	

Definition at line 80 of file INA3221.h.

8.57.3 Variable Documentation

8.57.3.1 INA3221_CH_NUM

```
const int INA3221_CH_NUM = 3
```

Number of channels in INA3221.

Definition at line 30 of file INA3221.h.

8.57.3.2 SHUNT_VOLTAGE_LSB_UV

```
const int SHUNT_VOLTAGE_LSB_UV = 5
```

LSB value for shunt voltage measurements in microvolts.

Definition at line 32 of file INA3221.h.

8.58 INA3221.h

Go to the documentation of this file.

```
#ifndef BEASTDEVICES_INA3221_H
#define BEASTDEVICES_INA3221_H

#include <stdint.h>
#include <iostream>
#include <hardware/i2c.h>

typedef enum {
    INA3221_ADDR40_GND = 0b1000000, // A0 pin -> GND
    INA3221_ADDR41_VCC = 0b1000001, // A0 pin -> VCC
    INA3221_ADDR42_SDA = 0b1000010, // A0 pin -> SDA
    INA3221_ADDR43_SCL = 0b1000011  // A0 pin -> SCL
} ina3221_addr_t;

typedef enum {
    INA3221_CH1 = 0,
    INA3221_CH2,
    INA3221_CH3,
} ina3221_ch_t;

const int INA3221_CH_NUM = 3;
const int SHUNT_VOLTAGE_LSB_UV = 5;


typedef enum {
    INA3221_REG_CONF = 0,
    INA3221_REG_CH1_SHUNTV,
    INA3221_REG_CH1_BUSV,
    INA3221_REG_CH2_SHUNTV,
    INA3221_REG_CH2_BUSV,
    INA3221_REG_CH3_SHUNTV,
    INA3221_REG_CH3_BUSV,
    INA3221_REG_CH1_CRIT_ALERT_LIM,
    INA3221_REG_CH1_WARNING_ALERT_LIM,
    INA3221_REG_CH2_CRIT_ALERT_LIM,
    INA3221_REG_CH2_WARNING_ALERT_LIM,
    INA3221_REG_CH3_CRIT_ALERT_LIM,
    INA3221_REG_CH3_WARNING_ALERT_LIM,
    INA3221_REG_SHUNTV_SUM,
    INA3221_REG_SHUNTV_SUM_LIM,
    INA3221_REG_MASK_ENABLE,
    INA3221_REG_PWR_VALID_HI_LIM,
    INA3221_REG_PWR_VALID_LO_LIM,
    INA3221_REG_MANUF_ID = 0xFE,
    INA3221_REG_DIE_ID = 0xFF
} ina3221_reg_t;

typedef enum {
    INA3221_REG_CONF_CT_140US = 0,
    INA3221_REG_CONF_CT_204US,
    INA3221_REG_CONF_CT_332US,
    INA3221_REG_CONF_CT_588US,
    INA3221_REG_CONF_CT_1100US,
    INA3221_REG_CONF_CT_2116US,
    INA3221_REG_CONF_CT_4156US,
    INA3221_REG_CONF_CT_8244US
} ina3221_conv_time_t;

typedef enum {
    INA3221_REG_CONF_AVG_1 = 0,
    INA3221_REG_CONF_AVG_4,
    INA3221_REG_CONF_AVG_16,
    INA3221_REG_CONF_AVG_64,
    INA3221_REG_CONF_AVG_128,
    INA3221_REG_CONF_AVG_256,
    INA3221_REG_CONF_AVG_512,
    INA3221_REG_CONF_AVG_1024
} ina3221_avg_mode_t;

class INA3221 {

    typedef struct {
        uint16_t mode_shunt_en:1;
        uint16_t mode_bus_en:1;
        uint16_t mode_continious_en:1;
        uint16_t shunt_conv_time:3;
        uint16_t bus_conv_time:3;
        uint16_t avg_mode:3;
        uint16_t ch3_en:1;
        uint16_t ch2_en:1;
        uint16_t ch1_en:1;
        uint16_t reset:1;
```

```
    } conf_reg_t;

    typedef struct {
        uint16_t conv_ready:1;
        uint16_t timing_ctrl_alert:1;
        uint16_t pwr_valid_alert:1;
        uint16_t warn_alert_ch3:1;
        uint16_t warn_alert_ch2:1;
        uint16_t warn_alert_ch1:1;
        uint16_t shunt_sum_alert:1;
        uint16_t crit_alert_ch3:1;
        uint16_t crit_alert_ch2:1;
        uint16_t crit_alert_ch1:1;
        uint16_t crit_alert_latch_en:1;
        uint16_t warn_alert_latch_en:1;
        uint16_t shunt_sum_en_ch3:1;
        uint16_t shunt_sum_en_ch2:1;
        uint16_t shunt_sum_en_ch1:1;
        uint16_t reserved:1;
    } masken_reg_t;

    // I2C address
    ina3221_addr_t _i2c_addr;
    i2c_inst_t* _i2c;

    // Shunt resistance in mOhm
    uint32_t _shuntRes[INA3221_CH_NUM];

    // Series filter resistance in Ohm
    uint32_t _filterRes[INA3221_CH_NUM];

    // Value of Mask/Enable register.
    masken_reg_t _masken_reg;

    // Reads 16 bytes from a register.
    void _read(ina3221_reg_t reg, uint16_t *val);

    // Writes 16 bytes to a register.
    void _write(ina3221_reg_t reg, uint16_t *val);

public:

    INA3221(ina3221_addr_t addr, i2c_inst_t* i2c);
    // Initializes INA3221
    bool begin();

    // Gets a register value.
    uint16_t read_register(ina3221_reg_t reg);

    // Resets INA3221
    void reset();

    // Sets operating mode to power-down
    void set_mode_power_down();

    // Sets operating mode to continious
    void set_mode_continuous();

    // Sets operating mode to triggered (single-shot)
    void set_mode_triggered();

    // Enables shunt-voltage measurement
    void set_shunt_measurement_enable();

    // Disables shunt-voltage mesurement
    void set_shunt_measurement_disable();

    // Enables bus-voltage measurement
    void set_bus_measurement_enable();

    // Disables bus-voltage measureement
    void set_bus_measurement_disable();

    // Sets averaging mode. Sets number of samples that are collected
    // and averaged togehter.
    void set_averaging_mode(ina3221_avg_mode_t mode);

    // Sets bus-voltage conversion time.
    void set_bus_conversion_time(ina3221_conv_time_t convTime);

    // Sets shunt-voltage conversion time.
    void set_shunt_conversion_time(ina3221_conv_time_t convTime);

    // Gets manufacturer ID.
    // Should read 0x5449.
    uint16_t get_manufacturer_id();

```

```
    // Gets die ID.
    // Should read 0x3220.
    uint16_t get_die_id();

    // Gets shunt voltage in uV.
    int32_t get_shunt_voltage(ina3221_ch_t channel);

    // Gets current in A.
    float get_current(ina3221_ch_t channel);

    float get_current_ma(ina3221_ch_t channel);

    // Gets bus voltage in V.
    float get_voltage(ina3221_ch_t channel);
};

#endif
```

8.59 lib/powerman/PowerManager.cpp File Reference

Implementation of the PowerManager class, which manages power-related functions.

```
#include "PowerManager.h"
#include <iostream>
#include <sstream>
#include "event_manager.h"
```

8.59.1 Detailed Description

Implementation of the PowerManager class, which manages power-related functions.

This file implements the PowerManager class, a singleton that provides methods for reading voltage and current values, configuring the INA3221 power monitor, and checking power alerts.

Definition in file PowerManager.cpp.

8.60 PowerManager.cpp

Go to the documentation of this file.

```


#include "PowerManager.h"
#include <iostream>
#include <sstream>
#include "event_manager.h"

PowerManager::PowerManager()
    : ina3221_(INA3221_ADDR40_GND, MAIN_I2C_PORT) {
    recursive_mutex_init(&powerman_mutex_);
}

PowerManager& PowerManager::get_instance() {
    static PowerManager instance;
    return instance;
}

bool PowerManager::initialize() {
    recursive_mutex_enter_blocking(&powerman_mutex_);
    initialized_ = ina3221_.begin();

    recursive_mutex_exit(&powerman_mutex_);
    return initialized_;
}
```

```
std::string PowerManager::read_device_ids() {
    if (!initialized_) return "noinit";
    recursive_mutex_enter_blocking(&powerman_mutex_);
    std::stringstream man_ss;
    man_ss « std::hex « ina3221_.get_manufacturer_id();
    std::string MAN = "MAN 0x" + man_ss.str();

    std::stringstream die_ss;
    die_ss « std::hex « ina3221_.get_die_id();
    std::string DIE = "DIE 0x" + die_ss.str();
    recursive_mutex_exit(&powerman_mutex_);
    return MAN + " - " + DIE;
}

float PowerManager::get_voltage_battery() {
    if (!initialized_) return 0.0f;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    float voltage = ina3221_.get_voltage(INA3221_CH1);
    recursive_mutex_exit(&powerman_mutex_);
    return voltage;
}

float PowerManager::get_voltage_5v() {
    if (!initialized_) return 0.0f;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    float voltage = ina3221_.get_voltage(INA3221_CH2);
    recursive_mutex_exit(&powerman_mutex_);
    return voltage;
}

float PowerManager::get_current_charge_usb() {
    if (!initialized_) return 0.0f;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    float current = ina3221_.get_current_ma(INA3221_CH1);
    recursive_mutex_exit(&powerman_mutex_);
    return current;
}

float PowerManager::get_current_draw() {
    if (!initialized_) return 0.0f;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    float current = ina3221_.get_current_ma(INA3221_CH2);
    recursive_mutex_exit(&powerman_mutex_);
    return current;
}

float PowerManager::get_current_charge_solar() {
    if (!initialized_) return 0.0f;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    float current = ina3221_.get_current_ma(INA3221_CH3);
    recursive_mutex_exit(&powerman_mutex_);
    return current;
}

float PowerManager::get_current_charge_total() {
    if (!initialized_) return 0.0f;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    float current = ina3221_.get_current_ma(INA3221_CH1) + ina3221_.get_current_ma(INA3221_CH3);
    recursive_mutex_exit(&powerman_mutex_);
    return current;
}

void PowerManager::configure(const std::map<std::string, std::string>& config) {
    if (!initialized_) return;
    recursive_mutex_enter_blocking(&powerman_mutex_);

    if (config.find("operating_mode") != config.end()) {
        if (config.at("operating_mode") == "continuous") {
            ina3221_.set_mode_continuous();
        }
    }

    if (config.find("averaging_mode") != config.end()) {
        int avg_mode = std::stoi(config.at("averaging_mode"));
        switch(avg_mode) {
            case 1:
                ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_1);
                break;
            case 4:
                ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_4);
                break;
            case 16:
                ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_16);
                break;
            default:
                ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_16);
```

```
        }
    }
    recursive_mutex_exit(&powerman_mutex_);
}

bool PowerManager::is_charging_solar() {
    if (!initialized_) return false;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    bool active = get_current_charge_solar() > SOLAR_CURRENT_THRESHOLD;
    recursive_mutex_exit(&powerman_mutex_);
    return active;
}

bool PowerManager::is_charging_usb() {
    if (!initialized_) return false;
    recursive_mutex_enter_blocking(&powerman_mutex_);
    bool connected = get_current_charge_usb() > USB_CURRENT_THRESHOLD;
    recursive_mutex_exit(&powerman_mutex_);
    return connected;
}

```

8.61 lib/powerman/PowerManager.h File Reference

Header file for the PowerManager class, which manages power-related functions.

```
#include "INA3221/INA3221.h"
#include <map>
#include <string>
#include <hardware/i2c.h>
#include "pico/stdlib.h"
#include "pico/mutex.h"
```

Classes

- class PowerManager

  Manages power-related functions.

8.61.1 Detailed Description

Header file for the PowerManager class, which manages power-related functions.

This file defines the PowerManager class, a singleton that provides methods for reading voltage and current values, configuring the INA3221 power monitor, and checking power alerts.

Definition in file PowerManager.h.

8.62 PowerManager.h

Go to the documentation of this file.

```
#ifndef POWER_MANAGER_H
#define POWER_MANAGER_H

#include "INA3221/INA3221.h"
#include <map>
#include <string>
#include <hardware/i2c.h>
#include "pico/stdlib.h"
#include "pico/mutex.h"

class PowerManager {
public:
    PowerManager(i2c_inst_t* i2c);

    static PowerManager& get_instance();

    bool initialize();

    std::string read_device_ids();

    float get_current_charge_solar();

    float get_current_charge_usb();

    float get_current_charge_total();

    float get_current_draw();

    float get_voltage_battery();

    float get_voltage_5v();

    void configure(const std::map<std::string, std::string>& config);

    bool is_charging_solar();

    bool is_charging_usb();


    static constexpr float SOLAR_CURRENT_THRESHOLD = 50.0f;  // mA
    static constexpr float USB_CURRENT_THRESHOLD = 50.0f;    // mA
    static constexpr float BATTERY_LOW_THRESHOLD = 2.8f;     // V
    static constexpr float BATTERY_FULL_THRESHOLD = 4.2f; // V

private:
    INA3221 ina3221_;
    bool initialized_;
    recursive_mutex_t powerman_mutex_;
    bool charging_solar_active_ = false;
    bool charging_usb_active_ = false;

    PowerManager();

    PowerManager(const PowerManager&) = delete;
    PowerManager& operator=(const PowerManager&) = delete;
};

 #endif // POWER_MANAGER_H
```

8.63 lib/sensors/BH1750/BH1750.cpp File Reference

Implementation of the BH1750 light sensor class.

```
#include "BH1750.h"
#include "pico/stdlib.h"
#include <stdio.h>
#include <iostream>
```

8.63.1 Detailed Description

Implementation of the BH1750 light sensor class.

This file contains the implementation of the BH1750 class, which provides an interface to the BH1750 digital light sensor using the I2C communication protocol.

Definition in file BH1750.cpp.

8.64 BH1750.cpp

Go to the documentation of this file.

```
#include "BH1750.h"
#include "pico/stdlib.h"
#include <stdio.h>
#include <iostream>

BH1750::BH1750(i2c_inst_t* i2c, uint8_t addr) : _i2c_addr(addr), i2c_port_(i2c) {}

bool BH1750::begin(Mode mode) {
    write8(static_cast<uint8_t>(Mode::POWER_ON));
    write8(static_cast<uint8_t>(Mode::RESET));
    bool config_status = configure(mode);

    return config_status;
}

bool BH1750::configure(Mode mode) {
    uint8_t modeVal = static_cast<uint8_t>(mode);
    switch (mode) {
        case Mode::UNCONFIGURED_POWER_DOWN:
        case Mode::POWER_ON:
        case Mode::RESET:
        case Mode::CONTINUOUS_HIGH_RES_MODE:
        case Mode::CONTINUOUS_HIGH_RES_MODE_2:
        case Mode::CONTINUOUS_LOW_RES_MODE:
        case Mode::ONE_TIME_HIGH_RES_MODE:
        case Mode::ONE_TIME_HIGH_RES_MODE_2:
        case Mode::ONE_TIME_LOW_RES_MODE:
            write8(modeVal);
            sleep_ms(10);
            return true;
        default:
            return false;
    }
}

float BH1750::get_light_level() {
    uint8_t buffer[2];
    i2c_read_blocking(i2c_port_, _i2c_addr, buffer, 2, false);
    uint16_t level = (buffer[0] « 8) | buffer[1];

    float lux = static_cast<float>(level) / 1.2f;
    return lux;
}

void BH1750::write8(uint8_t data) {
    uint8_t buf[1] = {data};
    i2c_write_blocking(i2c_port_, _i2c_addr, buf, 1, false);
}
```

8.65 lib/sensors/BH1750/BH1750.h File Reference

Header file for the BH1750 light sensor class.

```
#include "hardware/i2c.h"
```

Classes

- class BH1750

  Class to interface with the BH1750 light sensor.

Macros

- #define _BH1750_DEVICE_ID 0xE1

  Correct content of WHO_AM_I register (not actually used in this driver).
- #define _BH1750_MTREG_MIN 31

  Minimum value for the MTREG register.
- #define _BH1750_MTREG_MAX 254

  Maximum value for the MTREG register.
- #define _BH1750_DEFAULT_MTREG 69

  Default value for the MTREG register.

8.65.1 Detailed Description

Header file for the BH1750 light sensor class.

This class provides an interface to the BH1750 digital light sensor using the I2C communication protocol.

Definition in file BH1750.h.

8.66 BH1750.h

Go to the documentation of this file.

```
#ifndef __BH1750_H__
#define __BH1750_H__

#include "hardware/i2c.h"

#define _BH1750_DEVICE_ID 0xE1

#define _BH1750_MTREG_MIN 31

#define _BH1750_MTREG_MAX 254

#define _BH1750_DEFAULT_MTREG 69

class BH1750 {
public:
    enum class Mode : uint8_t {
        UNCONFIGURED_POWER_DOWN = 0x00,
        POWER_ON = 0x01,
        RESET = 0x07,
        CONTINUOUS_HIGH_RES_MODE = 0x10,
        CONTINUOUS_HIGH_RES_MODE_2 = 0x11,
        CONTINUOUS_LOW_RES_MODE = 0x13,
        ONE_TIME_HIGH_RES_MODE = 0x20,
        ONE_TIME_HIGH_RES_MODE_2 = 0x21,
        ONE_TIME_LOW_RES_MODE = 0x23
    };

    BH1750(i2c_inst_t* i2c, uint8_t addr = 0x23);

    bool begin(Mode mode = Mode::CONTINUOUS_HIGH_RES_MODE);

```

```
    bool configure(Mode mode);

    float get_light_level();

private:
    void write8(uint8_t data);

    uint8_t _i2c_addr;
    i2c_inst_t* i2c_port_;
};

#endif // __BH1750_H__
```

8.67 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference

```
#include "BH1750_WRAPPER.h"
#include <string>
#include <iostream>
```

8.68 BH1750_WRAPPER.cpp

Go to the documentation of this file.

```
#include "BH1750_WRAPPER.h"
#include <string>
#include <iostream>

BH1750Wrapper::BH1750Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {
    sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
}

bool BH1750Wrapper::init() {
    initialized_ = sensor_.begin();
    return initialized_;
}

float BH1750Wrapper::read_data(SensorDataTypeIdentifier type) {
    if (type == SensorDataTypeIdentifier::LIGHT_LEVEL) {
        return sensor_.get_light_level();
    }
    return 0.0f;
}

bool BH1750Wrapper::is_initialized() const {
    return initialized_;
}

SensorType BH1750Wrapper::get_type() const {
    return SensorType::LIGHT;
}

bool BH1750Wrapper::configure(const std::map<std::string, std::string>& config) {
    for (const auto& [key, value] : config) {
        if (key == "measurement_mode") {
            if (value == "continuously_high_resolution") {
                sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
            }
            else if (value == "continuously_high_resolution_2") {
                sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE_2);
            }
            else if (value == "continuously_low_resolution") {
                sensor_.configure(BH1750::Mode::CONTINUOUS_LOW_RES_MODE);
            }
            else if (value == "one_time_high_resolution") {
                sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE);
            }
            else if (value == "one_time_high_resolution_2") {
                sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE_2);
            }
            else if (value == "one_time_low_resolution") {
                sensor_.configure(BH1750::Mode::ONE_TIME_LOW_RES_MODE);
            }
            else {
```

```
                std::cerr « "[BH1750Wrapper] Unknown measurement_mode value: " « value « std::endl;
                return false;
            }
        }
        else {
            std::cerr « "[BH1750Wrapper] Unknown configuration key: " « key « std::endl;
            return false;
        }
    }
    return true;
}
```

8.69 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "BH1750.h"
#include <map>
#include <string>
```

Classes

- class BH1750Wrapper

8.70 BH1750_WRAPPER.h

Go to the documentation of this file.

```
#ifndef BH1750_WRAPPER_H
#define BH1750_WRAPPER_H

#include "ISensor.h"
#include "BH1750.h"
#include <map>
#include <string>

class BH1750Wrapper : public ISensor {
private:
    BH1750 sensor_;
    bool initialized_ = false;

public:
    BH1750Wrapper(i2c_inst_t* i2c);
    BH1750Wrapper();
    int get_i2c_addr();
    bool init() override;
    float read_data(SensorDataTypeIdentifier type) override;
    bool is_initialized() const override;
    SensorType get_type() const override;

    bool configure(const std::map<std::string, std::string>& config);

    uint8_t get_address() const override {
        return 0x23;
    }
};

#endif // BH1750_WRAPPER_H
```

8.71 lib/sensors/BME280/BME280.cpp File Reference

Implementation of the BME280 environmental sensor class.

```
#include "BME280.h"
#include <iomanip>
#include <vector>
#include <algorithm>
#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdlib.h"
#include "utils.h"
```

8.71.1 Detailed Description

Implementation of the BME280 environmental sensor class.

This file contains the implementation of the BME280 class, which provides an interface to the BME280 temperature, pressure, and humidity sensor using the I2C communication protocol.

Definition in file BME280.cpp.

8.72 BME280.cpp

Go to the documentation of this file.

```


#include "BME280.h"
#include <iomanip>
#include <vector>
#include <algorithm>
#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdlib.h"
#include "utils.h"

BME280::BME280(i2c_inst_t* i2cPort, uint8_t address)
    : i2c_port(i2cPort), device_addr(address), calib_params{}, initialized_(false), t_fine(0) {
}

bool BME280::init() {
    if (!i2c_port) {
        uart_print("BME280 I2C port not initialized.", VerbosityLevel::ERROR);
        return false;
    }

    // Check device ID to confirm it's a BME280
    uint8_t chip_id;
    if (!read_register(0xD0, &chip_id)) {
        uart_print("Failed to read chip ID from BME280.", VerbosityLevel::ERROR);
        return false;
    }

    if (chip_id != 0x60) {
        uart_print("Invalid BME280 chip ID.", VerbosityLevel::ERROR);
        return false;
    }

    // Configure sensor
    if (!configure_sensor()) {
        uart_print("Failed to configure BME280 sensor.", VerbosityLevel::ERROR);
        return false;
    }

    // Retrieve calibration parameters
```

```
    if (!get_calibration_parameters()) {
        uart_print("Failed to get calibration parameters from BME280.", VerbosityLevel::ERROR);
        return false;
    }

    initialized_ = true;
    uart_print("BME280 initialized.", VerbosityLevel::INFO);
    return true;
}

void BME280::reset() {
    write_register(REG_RESET, 0xB6);
    sleep_ms(10); // Wait for reset to complete
}

bool BME280::read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity) {
    if (!initialized_) {
        uart_print("BME280 not initialized.", VerbosityLevel::ERROR);
        return false;
    }

    // Define the starting register address
    uint8_t start_reg = REG_PRESSURE_MSB;
    // Total bytes to read: 3 (pressure) + 3 (temperature) + 2 (humidity) = 8
    uint8_t buf[8] = {0};

    // Write the starting register address
    if (!write_register(start_reg, 1)) {
        uart_print("Failed to write to BME280.", VerbosityLevel::ERROR);
        return false;
    }

    // Read data
    int ret = i2c_read_blocking(i2c_port, device_addr, buf, 8, false);
    if (ret != 8) {
        uart_print("Failed to read from BME280.", VerbosityLevel::ERROR);
        return false;
    }

    // Combine bytes to form raw values
    *pressure = ((int32_t)buf[0] « 12) | ((int32_t)buf[1] « 4) | ((int32_t)(buf[2] » 4));
    *temperature = ((int32_t)buf[3] « 12) | ((int32_t)buf[4] « 4) | ((int32_t)(buf[5] » 4));
    *humidity = ((int32_t)buf[6] « 8) | (int32_t)buf[7];

    return true;
}

float BME280::convert_temperature(int32_t temp_raw) const {
    int32_t var1, var2;
    var1 = ((((temp_raw » 3) - ((int32_t)calib_params.dig_t1 « 1))) * ((int32_t)calib_params.dig_t2)) » 11;
    var2 = (((((temp_raw » 4) - ((int32_t)calib_params.dig_t1)) * ((temp_raw » 4) - ((int32_t)calib_params.dig_t1))) » 12) * ((int32_t)calib_params.dig_t3)) » 14;
    t_fine = var1 + var2;
    float T = (t_fine * 5 + 128) » 8;
    return T / 100.0f;
}

float BME280::convert_pressure(int32_t pressure_raw) const {
    int64_t var1, var2, p;
    var1 = ((int64_t)t_fine) - 128000;
    var2 = var1 * var1 * (int64_t)calib_params.dig_p6;
    var2 = var2 + ((var1 * (int64_t)calib_params.dig_p5) « 17);
    var2 = var2 + (((int64_t)calib_params.dig_p4) « 35);
    var1 = ((var1 * var1 * (int64_t)calib_params.dig_p3) » 8) + ((var1 * (int64_t)calib_params.dig_p2) « 12);
    var1 = (((((int64_t)1 « 47) + var1)) * ((int64_t)calib_params.dig_p1) » 33;

    if (var1 == 0) {
        return 0.0f; // avoid exception caused by division by zero
    }
    p = 1048576 - pressure_raw;
    p = (((p « 31) - var2) * 3125) / var1;
    var1 = (((int64_t)calib_params.dig_p9) * (p » 13) * (p » 13)) » 25;
    var2 = (((int64_t)calib_params.dig_p8) * p) » 19;

    p = ((p + var1 + var2) » 8) + (((int64_t)calib_params.dig_p7) « 4);
    return (float)p / 25600.0f; // in hPa
}

float BME280::convert_humidity(int32_t humidity_raw) const {
    int32_t v_x1_u32r;
    v_x1_u32r = t_fine - 76800;
    v_x1_u32r = (((((humidity_raw « 14) - ((int32_t)calib_params.dig_h4 « 20) - ((int32_t)calib_params.dig_h5 * v_x1_u32r)) + 16384) » 15) *
                  ((((((v_x1_u32r * (int32_t)calib_params.dig_h6) » 10) * (((v_x1_u32r * (int32_t)calib_params.dig_h3) » 11) + 32768)) » 10) + 2097152) *
```

```
                    (int32_t)calib_params.dig_h2 + 8192) » 14));
    v_x1_u32r = std::max(v_x1_u32r, (int32_t)0);
    v_x1_u32r = std::min(v_x1_u32r, (int32_t)419430400);
    float h = v_x1_u32r » 12;
    return h / 1024.0f;
}

bool BME280::get_calibration_parameters() {
    // Read temperature and pressure calibration data (0x88 to 0xA1)
    uint8_t calib_data[NUM_CALIB_PARAMS];
    if (!read_register(REG_DIG_T1_LSB, calib_data, NUM_CALIB_PARAMS)) {
        uart_print("Failed to read calibration data from BME280.", VerbosityLevel::ERROR);
        return false;
    }

    // Parse temperature calibration data
    calib_params.dig_t1 = (uint16_t)(calib_data[1] « 8 | calib_data[0]);
    calib_params.dig_t2 = (int16_t)(calib_data[3] « 8 | calib_data[2]);
    calib_params.dig_t3 = (int16_t)(calib_data[5] « 8 | calib_data[4]);

    // Parse pressure calibration data
    calib_params.dig_p1 = (uint16_t)(calib_data[7] « 8 | calib_data[6]);
    calib_params.dig_p2 = (int16_t)(calib_data[9] « 8 | calib_data[8]);
    calib_params.dig_p3 = (int16_t)(calib_data[11] « 8 | calib_data[10]);
    calib_params.dig_p4 = (int16_t)(calib_data[13] « 8 | calib_data[12]);
    calib_params.dig_p5 = (int16_t)(calib_data[15] « 8 | calib_data[14]);
    calib_params.dig_p6 = (int16_t)(calib_data[17] « 8 | calib_data[16]);
    calib_params.dig_p7 = (int16_t)(calib_data[19] « 8 | calib_data[18]);
    calib_params.dig_p8 = (int16_t)(calib_data[21] « 8 | calib_data[20]);
    calib_params.dig_p9 = (int16_t)(calib_data[23] « 8 | calib_data[22]);

    calib_params.dig_h1 = calib_data[25];

    // Read humidity calibration data (0xE1 to 0xE7)
    uint8_t hum_calib_data[NUM_HUM_CALIB_PARAMS];
    if (!read_register(REG_DIG_H2, hum_calib_data, NUM_HUM_CALIB_PARAMS)) {
        uart_print("Failed to read humidity calibration data from BME280.", VerbosityLevel::ERROR);
        return false;
    }

    // Parse humidity calibration data
    calib_params.dig_h2 = (int16_t)(hum_calib_data[1] « 8 | hum_calib_data[0]);
    calib_params.dig_h3 = hum_calib_data[2];
    calib_params.dig_h4 = (int16_t)((hum_calib_data[3] « 4) | (hum_calib_data[4] & 0x0F));
    calib_params.dig_h5 = (int16_t)((hum_calib_data[5] « 4) | (hum_calib_data[4] » 4));
    calib_params.dig_h6 = (int8_t)hum_calib_data[6];

    return true;
}

bool BME280::configure_sensor() {
    // Set humidity oversampling (must be set before ctrl_meas)
    if (!write_register(REG_CTRL_HUM, HUMIDITY_OVERSAMPLING)) {
        uart_print("Failed to write CTRL_HUM to BME280.", VerbosityLevel::ERROR);
        return false;
    }

    // Write config register
    if (!write_register(REG_CONFIG, 0x00)) {
        uart_print("Failed to write CONFIG to BME280.", VerbosityLevel::ERROR);
        return false;
    }

    // Write ctrl_meas register
    if (!write_register(REG_CTRL_MEAS, NORMAL_MODE)) {
        uart_print("Failed to write CTRL_MEAS to BME280.", VerbosityLevel::ERROR);
        return false;
    }

    return true;
}

bool BME280::write_register(uint8_t reg, uint8_t value) {
    uint8_t buf[2] = {reg, value};
    int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
    return (ret == 2);
}

bool BME280::read_register(uint8_t reg, uint8_t* data, size_t len) {
    int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
    if (ret != 1) {
        return false;
    }
    ret = i2c_read_blocking(i2c_port, device_addr, data, len, false);
    return (static_cast<size_t>(ret) == len);
}
```

```
bool BME280::read_register(uint8_t reg, uint8_t* data) {
    return read_register(reg, data, 1);
}
```

8.73 lib/sensors/BME280/BME280.h File Reference

Header file for the BME280 environmental sensor class.

```
#include <cstdint>
#include <iostream>
#include "hardware/i2c.h"
```

Classes

- struct BME280CalibParam

  Structure to hold the BME280 calibration parameters.
- class BME280

  Class to interface with the BME280 environmental sensor.

8.73.1 Detailed Description

Header file for the BME280 environmental sensor class.

This class provides an interface to the BME280 temperature, pressure, and humidity sensor using the I2C communication protocol. It includes functions for initialization, reading raw sensor data, converting raw data to physical units, and configuring the sensor's operating mode.

Definition in file BME280.h.

8.74 BME280.h

Go to the documentation of this file.

```

#ifndef BME280_H
#define BME280_H

#include <cstdint>
#include <iostream>
#include "hardware/i2c.h"

struct BME280CalibParam {
    uint16_t dig_t1;
    int16_t  dig_t2;
    int16_t  dig_t3;

    uint16_t dig_p1;
    int16_t  dig_p2;
    int16_t  dig_p3;
    int16_t  dig_p4;
    int16_t  dig_p5;
    int16_t  dig_p6;
    int16_t  dig_p7;
    int16_t  dig_p8;
    int16_t  dig_p9;

    uint8_t  dig_h1;
```

```
    int16_t  dig_h2;
    uint8_t  dig_h3;
    int16_t  dig_h4;
    int16_t  dig_h5;
    int8_t   dig_h6;
};
class BME280 {
public:
    enum {
        ADDR_SDO_LOW = 0x76,
        ADDR_SDO_HIGH = 0x77
    };
    enum class Oversampling : uint8_t {
        OSR_X0 = 0x00,
        OSR_X1 = 0x01,
        OSR_X2 = 0x02,
        OSR_X4 = 0x03,
        OSR_X8 = 0x04,
        OSR_X16 = 0x05
    };
    BME280(i2c_inst_t* i2cPort, uint8_t address = ADDR_SDO_LOW);
    bool init();
    void reset();
    bool read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity);
    float convert_temperature(int32_t temp_raw) const;
    float convert_pressure(int32_t pressure_raw) const;
    float convert_humidity(int32_t humidity_raw) const;
private:
    bool write_register(uint8_t reg, uint8_t value);
    bool read_register(uint8_t reg, uint8_t* data);
    bool read_register(uint8_t reg, uint8_t* data, size_t len);
    bool configure_sensor();
    bool get_calibration_parameters();
    i2c_inst_t* i2c_port;
    uint8_t device_addr;
    BME280CalibParam calib_params;
    bool initialized_;
    mutable int32_t t_fine;
    enum {
        REG_CONFIG            = 0xF5,
        REG_CTRL_MEAS         = 0xF4,
        REG_CTRL_HUM          = 0xF2,
        REG_RESET             = 0xE0,
        REG_PRESSURE_MSB      = 0xF7,
        REG_TEMPERATURE_MSB   = 0xFA,
        REG_HUMIDITY_MSB      = 0xFD,
        // Calibration Registers
        REG_DIG_T1_LSB        = 0x88,
        REG_DIG_T1_MSB        = 0x89,
        REG_DIG_T2_LSB        = 0x8A,
        REG_DIG_T2_MSB        = 0x8B,
        REG_DIG_T3_LSB        = 0x8C,
        REG_DIG_T3_MSB        = 0x8D,
        REG_DIG_P1_LSB        = 0x8E,
        REG_DIG_P1_MSB        = 0x8F,
        REG_DIG_P2_LSB        = 0x90,
        REG_DIG_P2_MSB        = 0x91,
        REG_DIG_P3_LSB        = 0x92,
        REG_DIG_P3_MSB        = 0x93,
        REG_DIG_P4_LSB        = 0x94,
        REG_DIG_P4_MSB        = 0x95,
        REG_DIG_P5_LSB        = 0x96,
        REG_DIG_P5_MSB        = 0x97,
        REG_DIG_P6_LSB        = 0x98,
        REG_DIG_P6_MSB        = 0x99,
```

```
        REG_DIG_P7_LSB        = 0x9A,
        REG_DIG_P7_MSB        = 0x9B,
        REG_DIG_P8_LSB        = 0x9C,
        REG_DIG_P8_MSB        = 0x9D,
        REG_DIG_P9_LSB        = 0x9E,
        REG_DIG_P9_MSB        = 0x9F,

        // Humidity Calibration Registers
        REG_DIG_H1            = 0xA1,
        REG_DIG_H2            = 0xE1,
        REG_DIG_H3            = 0xE3,
        REG_DIG_H4            = 0xE4,
        REG_DIG_H5            = 0xE5,
        REG_DIG_H6            = 0xE7
    };

    enum {
        HUMIDITY_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
        TEMPERATURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
        PRESSURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
        NORMAL_MODE = 0xB7
    };

    enum {
        NUM_CALIB_PARAMS = 26,
        NUM_HUM_CALIB_PARAMS = 7
    };
};

#endif // BME280_H
```

8.75 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference

```
#include "BME280_WRAPPER.h"
```

8.76 BME280_WRAPPER.cpp

Go to the documentation of this file.

```
#include "BME280_WRAPPER.h"

BME280Wrapper::BME280Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {}

bool BME280Wrapper::init() {
    initialized_ = sensor_.init();
    return initialized_;
}

float BME280Wrapper::read_data(SensorDataTypeIdentifier type) {
    int32_t temp_raw, pressure_raw, humidity_raw;
    sensor_.read_raw_all(&temp_raw, &pressure_raw, &humidity_raw);

    switch(type) {
        case SensorDataTypeIdentifier::TEMPERATURE:
            return sensor_.convert_temperature(temp_raw);
        case SensorDataTypeIdentifier::PRESSURE:
            return sensor_.convert_pressure(pressure_raw);
        case SensorDataTypeIdentifier::HUMIDITY:
            return sensor_.convert_humidity(humidity_raw);
        default:
            return 0.0f;
    }
}

bool BME280Wrapper::is_initialized() const {
    return initialized_;
}

SensorType BME280Wrapper::get_type() const {
    return SensorType::ENVIRONMENT;
}

bool BME280Wrapper::configure([[maybe_unused]] const std::map<std::string, std::string>& config) {
    return true;
}
```

8.77 lib/sensors/BME280/BME280_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "BME280.h"
```

Classes

- class BME280Wrapper

8.78 BME280_WRAPPER.h

Go to the documentation of this file.

```
// BME280_WRAPPER.h
#ifndef BME280_WRAPPER_H
#define BME280_WRAPPER_H

#include "ISensor.h"
#include "BME280.h"

class BME280Wrapper : public ISensor {
private:
    BME280 sensor_;
    bool initialized_ = false;

public:
    BME280Wrapper(i2c_inst_t* i2c);

    bool init() override;
    float read_data(SensorDataTypeIdentifier type) override;
    bool is_initialized() const override;
    SensorType get_type() const override;
    bool configure(const std::map<std::string, std::string>& config) override;

    uint8_t get_address() const override {
        return 0x76;
    }

};

#endif // BME280_WRAPPER_H
```

8.79 lib/sensors/ISensor.cpp File Reference

Implementation of the ISensor interface and SensorWrapper class.

```
#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/utils.h"
```

8.79.1 Detailed Description

Implementation of the ISensor interface and SensorWrapper class.

This file implements the ISensor interface and SensorWrapper class, which provide a common interface for interacting with different types of sensors.

Definition in file ISensor.cpp.

8.80 ISensor.cpp

Go to the documentation of this file.

```
#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/utils.h"

bool SensorWrapper::sensor_init(SensorType type, i2c_inst_t* i2c) {
    switch (type) {
    case SensorType::LIGHT:
        sensors[type] = new BH1750Wrapper(i2c);
        break;
    case SensorType::ENVIRONMENT:
        sensors[type] = new BME280Wrapper(i2c);
        break;
    default:
        return false;
    }
    return sensors[type]->init();
}

bool SensorWrapper::sensor_configure(SensorType type, const std::map<std::string, std::string>& config) {
    if (sensors.find(type) == sensors.end()) {
        return false;
    }
    return sensors[type]->configure(config);
}

float SensorWrapper::sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType) {
    if (sensors.find(sensorType) == sensors.end()) {
        return -1.0f;
    }
    return sensors[sensorType]->read_data(dataType);
}

ISensor* SensorWrapper::get_sensor(SensorType type) {
    return sensors[type];
}

std::vector<std::pair<SensorType, uint8_t» SensorWrapper::scan_connected_sensors(i2c_inst_t* i2c) {
    std::vector<std::pair<SensorType, uint8_t» connectedSensors;

    // Scan for BH1750 (Light Sensor)
    BH1750Wrapper lightSensor(i2c);
    if (lightSensor.init()) {
        connectedSensors.push_back(std::make_pair(SensorType::LIGHT, lightSensor.get_address()));
    }

    // Scan for BME280 (Environment Sensor)
    BME280Wrapper environmentSensor(i2c);
    if (environmentSensor.init()) {
        connectedSensors.push_back(std::make_pair(SensorType::ENVIRONMENT, environmentSensor.get_address()));
    }

    return connectedSensors;
}

std::vector<std::pair<SensorType, uint8_t» SensorWrapper::get_available_sensors() {
    std::vector<std::pair<SensorType, uint8_t» availableSensors;
    for (const auto& sensorPair : sensors) {
        availableSensors.push_back(std::make_pair(sensorPair.first, sensorPair.second->get_address()));
    }
    return availableSensors;
}
```

8.81 lib/sensors/ISensor.h File Reference

Header file for the ISensor interface and SensorWrapper class.

```
#include <map>
#include <string>
```

```
#include <vector>
#include <utility>
#include "hardware/i2c.h"
```

Classes

- class ISensor

    Abstract base class for sensors.
- class SensorWrapper

    Manages a collection of sensors.

Enumerations

- enum class SensorType : uint8_t { SensorType::NONE = 0x00 , SensorType::LIGHT = 0x01 , SensorType::ENVIRONMENT = 0x02 }

    Enumeration of sensor types.
- enum class SensorDataTypeIdentifier : uint8_t {
SensorDataTypeIdentifier::NONE = 0x00 , SensorDataTypeIdentifier::LIGHT_LEVEL = 0x01 , SensorDataTypeIdentifier::TEMPE
= 0x02 , SensorDataTypeIdentifier::HUMIDITY = 0x03 ,
SensorDataTypeIdentifier::PRESSURE = 0x04 }

    Enumeration of sensor data type identifiers.

8.81.1 Detailed Description

Header file for the ISensor interface and SensorWrapper class.

This file defines the ISensor interface, which provides a common interface for interacting with different types of sensors. It also defines the SensorWrapper class, which manages a collection of sensors and provides methods for initializing, configuring, and reading data from them.

Definition in file ISensor.h.

8.82 ISensor.h

Go to the documentation of this file.

```


#ifndef ISENSOR_H
#define ISENSOR_H

#include <map>
#include <string>
#include <vector>
#include <utility>
#include "hardware/i2c.h"

enum class SensorType : uint8_t {
    NONE = 0x00,
    LIGHT = 0x01,
    ENVIRONMENT = 0x02,
};

enum class SensorDataTypeIdentifier : uint8_t {
    NONE = 0x00,
    LIGHT_LEVEL = 0x01,
    TEMPERATURE = 0x02,
```

```
    HUMIDITY = 0x03,
    PRESSURE = 0x04,
};

class ISensor {
public:
    virtual ~ISensor() = default;

    virtual bool init() = 0;

    virtual float read_data(SensorDataTypeIdentifier type) = 0;

    virtual bool is_initialized() const = 0;

    virtual SensorType get_type() const = 0;

    virtual bool configure(const std::map<std::string, std::string>& config) = 0;

    virtual uint8_t get_address() const = 0;
};

class SensorWrapper {
public:
    static SensorWrapper& get_instance() {
        static SensorWrapper instance;
        return instance;
    }

    bool sensor_init(SensorType type, i2c_inst_t* i2c = nullptr);

    bool sensor_configure(SensorType type, const std::map<std::string, std::string>& config);

    float sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType);

    ISensor* get_sensor(SensorType type);

    std::vector<std::pair<SensorType, uint8_t» scan_connected_sensors(i2c_inst_t* i2c);

    std::vector<std::pair<SensorType, uint8_t» get_available_sensors();

private:
    std::map<SensorType, ISensor*> sensors;

    SensorWrapper() = default;
};

#endif
```

8.83 lib/storage/storage.cpp File Reference

Implements file system operations for the Kubisat firmware.

```
#include "storage.h"
#include "errno.h"
#include "utils.h"
#include "system_state_manager.h"
```

Functions

- bool fs_init (void)

  Initializes the file system on the SD card.
- bool fs_stop (void)

  Unmounts the file system from the SD card.

8.83.1 Detailed Description

Implements file system operations for the Kubisat firmware.

This file contains functions for initializing the file system, opening, writing, reading, and closing files.

Definition in file storage.cpp.

8.84 storage.cpp

Go to the documentation of this file.

```
#include "storage.h"
#include "errno.h"
#include "utils.h"
#include "system_state_manager.h"

bool fs_init(void) {
    SystemStateManager::get_instance().set_sd_card_mounted(false);
    uart_print("fs_init littlefs on SD card", VerbosityLevel::DEBUG);
    blockdevice_t *sd = blockdevice_sd_create(SD_SPI_PORT,
                                              SD_MOSI_PIN,
                                              SD_MISO_PIN,
                                              SD_SCK_PIN,
                                              SD_CS_PIN,
                                              24 * MHZ,
                                              false);
    filesystem_t *fat = filesystem_fat_create();

    std::string status_string;
    int err = fs_mount("/", fat, sd);
    if (err == -1) {
        status_string = "Formatting / with FAT";
        uart_print(status_string, VerbosityLevel::WARNING);
        err = fs_format(fat, sd);
        if (err == -1) {
            status_string = "fs_format error: " + std::string(strerror(errno));
            uart_print(status_string, VerbosityLevel::ERROR);
            return false;
        }
        err = fs_mount("/", fat, sd);
        if (err == -1) {
            status_string = "fs_mount error: " + std::string(strerror(errno));
            uart_print(status_string, VerbosityLevel::ERROR);
            return false;
        }
    }

    SystemStateManager::get_instance().set_sd_card_mounted(true);
    return true;
}

bool fs_stop(void) {
    int err = fs_unmount("/");
    if (err == -1) {
        uart_print("fs_unmount error", VerbosityLevel::ERROR);
        return false;
    }
    SystemStateManager::get_instance().set_sd_card_mounted(false);

    return true;
}
```

8.85 lib/storage/storage.h File Reference

Header file for file system operations on the Kubisat firmware.

```
#include <stdio.h>
#include <string.h>
#include <hardware/clocks.h>
#include <hardware/flash.h>
#include "blockdevice/flash.h"
#include "blockdevice/sd.h"
#include "filesystem/littlefs.h"
#include "filesystem/vfs.h"
#include "pin_config.h"
#include "lfs.h"
#include "filesystem/fat.h"
```

Functions

- bool fs_init (void)

  Initializes the file system on the SD card.
- bool fs_stop (void)

  Unmounts the file system from the SD card.

8.85.1 Detailed Description

Header file for file system operations on the Kubisat firmware.

This file defines functions for initializing, mounting, and unmounting the file system on the SD card.

Definition in file storage.h.

8.86 storage.h

Go to the documentation of this file.

```
#ifndef STORAGE_H
#define STORAGE_H

#include <stdio.h>
#include <string.h>
#include <hardware/clocks.h>
#include <hardware/flash.h>
#include "blockdevice/flash.h"
#include "blockdevice/sd.h"
#include "filesystem/littlefs.h"
#include "filesystem/vfs.h"
#include "pin_config.h"
#include "lfs.h"
#include "filesystem/fat.h"

bool fs_init(void);

bool fs_stop(void);

#endif
```

8.87 lib/system_state_manager.h File Reference

Manages the system state of the Kubisat firmware.

```
#include <mutex>
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
```

Classes

- class SystemStateManager

  Manages the system state of the Kubisat firmware.

8.87.1 Detailed Description

Manages the system state of the Kubisat firmware.

This class is a singleton that provides methods for getting and setting various system states, such as whether a bootloader reset is pending, whether GPS collection is paused, whether the SD card is mounted, and the UART verbosity level.

Definition in file system_state_manager.h.

8.88 system_state_manager.h

Go to the documentation of this file.

```


#ifndef SYSTEM_STATE_MANAGER_H
#define SYSTEM_STATE_MANAGER_H

#include <mutex>
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"

class SystemStateManager {
    private:
        bool pending_bootloader_reset;
        bool gps_collection_paused;
        bool sd_card_mounted;
        VerbosityLevel uart_verbosity;
        bool sd_card_init_status;
        bool radio_init_status;
        bool light_sensor_init_status;
        bool env_sensor_init_status;
        recursive_mutex_t mutex_;

        SystemStateManager() :
        pending_bootloader_reset(false),
        gps_collection_paused(false),
        sd_card_mounted(false),
        uart_verbosity(VerbosityLevel::DEBUG),
        sd_card_init_status(false),
        radio_init_status(false),
        light_sensor_init_status(false),
        env_sensor_init_status(false)
        {
            recursive_mutex_init(&mutex_);
        }

        SystemStateManager(const SystemStateManager&) = delete;
        SystemStateManager& operator=(const SystemStateManager&) = delete;

    public:
        static SystemStateManager& get_instance() {
            static SystemStateManager instance;
            return instance;
        }

        bool is_bootloader_reset_pending() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            bool result = pending_bootloader_reset;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }

        void set_bootloader_reset_pending(bool pending) {
            recursive_mutex_enter_blocking(&mutex_);
            pending_bootloader_reset = pending;
            recursive_mutex_exit(&mutex_);
        }

        bool is_gps_collection_paused() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            bool result = gps_collection_paused;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }
```

```
        void set_gps_collection_paused(bool paused) {
            recursive_mutex_enter_blocking(&mutex_);
            gps_collection_paused = paused;
            recursive_mutex_exit(&mutex_);
        }

        bool is_sd_card_mounted() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            bool result = sd_card_mounted;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }

        void set_sd_card_mounted(bool mounted) {
            recursive_mutex_enter_blocking(&mutex_);
            sd_card_mounted = mounted;
            recursive_mutex_exit(&mutex_);
        }

        VerbosityLevel get_uart_verbosity() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            VerbosityLevel result = uart_verbosity;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }

        void set_uart_verbosity(VerbosityLevel level) {
            recursive_mutex_enter_blocking(&mutex_);
            uart_verbosity = level;
            recursive_mutex_exit(&mutex_);
        }

        bool is_radio_init_ok() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            bool result = radio_init_status;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }

        void set_radio_init_ok(bool status) {
            recursive_mutex_enter_blocking(&mutex_);
            radio_init_status = status;
            recursive_mutex_exit(&mutex_);
        }

        bool is_light_sensor_init_ok() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            bool result = light_sensor_init_status;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }

        void set_light_sensor_init_ok(bool status) {
            recursive_mutex_enter_blocking(&mutex_);
            light_sensor_init_status = status;
            recursive_mutex_exit(&mutex_);
        }

        bool is_env_sensor_init_ok() const {
            recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
            bool result = env_sensor_init_status;
            recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
            return result;
        }

        void set_env_sensor_init_ok(bool status) {
            recursive_mutex_enter_blocking(&mutex_);
            env_sensor_init_status = status;
            recursive_mutex_exit(&mutex_);
        }
    };

#endif
```

8.89 lib/telemetry/telemetry_manager.cpp File Reference

Implementation of telemetry collection and storage functionality.

```
#include "telemetry_manager.h"
#include "utils.h"
```

```
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include "system_state_manager.h"
```

Macros

- #define TELEMETRY_CSV_PATH "/telemetry.csv"

    Path to the telemetry CSV file on storage media.
- #define SENSOR_DATA_CSV_PATH "/sensors.csv"

    Path to the sensor data CSV file on storage media.
- #define DEFAULT_SAMPLE_INTERVAL_MS 1000

    Default interval between telemetry samples in milliseconds (2 seconds)
- #define DEFAULT_FLUSH_THRESHOLD 10

    Default number of records to collect before flushing to storage.

8.89.1 Detailed Description

Implementation of telemetry collection and storage functionality.

Handles collecting, buffering, and persisting telemetry data from various satellite subsystems including power, sensors, and GPS

Definition in file telemetry_manager.cpp.

8.90 telemetry_manager.cpp

Go to the documentation of this file.

```

#include "telemetry_manager.h"
#include "utils.h"
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include "system_state_manager.h"

#define TELEMETRY_CSV_PATH "/telemetry.csv"

#define SENSOR_DATA_CSV_PATH "/sensors.csv"

#define DEFAULT_SAMPLE_INTERVAL_MS 1000

#define DEFAULT_FLUSH_THRESHOLD 10
```

```
TelemetryManager::TelemetryManager() {}

bool TelemetryManager::init() {
    mutex_init(&telemetry_mutex);
    if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
        uart_print("Telemetry system initialized (storage not available)", VerbosityLevel::WARNING);
        return false;
    }

    bool success = true;

    FILE* telemetry_file = fopen(TELEMETRY_CSV_PATH, "w");
    if (!telemetry_file) {
        telemetry_file = fopen(TELEMETRY_CSV_PATH, "w");
        if (telemetry_file) {
            fprintf(telemetry_file, "timestamp,build,battery_v,system_v,usb_ma,solar_ma,discharge_ma,"
                "gps_time,latitude,lat_dir,longitude,lon_dir,speed_knots,course_deg,date,"
                "fix_quality,satellites,altitude_m\n");
            fclose(telemetry_file);
            uart_print("Created new telemetry log", VerbosityLevel::INFO);
        }
        else {
            uart_print("Failed to create telemetry log", VerbosityLevel::ERROR);
            success = false;
        }
    }
    else {
        fclose(telemetry_file);
    }

    FILE* sensor_file = fopen(SENSOR_DATA_CSV_PATH, "w");
    if (!sensor_file) {
        sensor_file = fopen(SENSOR_DATA_CSV_PATH, "w");
        if (sensor_file) {
            fprintf(sensor_file, "timestamp,temperature,pressure,humidity,light\n");
            fclose(sensor_file);
            uart_print("Created new sensor data log", VerbosityLevel::INFO);
        }
        else {
            uart_print("Failed to create sensor data log", VerbosityLevel::ERROR);
            success = false;
        }
    }
    else {
        fclose(sensor_file);
    }

    return success;
}


void TelemetryManager::collect_power_telemetry(TelemetryRecord& record) {
    record.battery_voltage = PowerManager::get_instance().get_voltage_battery();
    record.system_voltage = PowerManager::get_instance().get_voltage_5v();
    record.charge_current_usb = PowerManager::get_instance().get_current_charge_usb();
    record.charge_current_solar = PowerManager::get_instance().get_current_charge_solar();
    record.discharge_current = PowerManager::get_instance().get_current_draw();
}

void TelemetryManager::emit_power_events(float battery_voltage, float charge_current_usb, float
    charge_current_solar) {
    static bool usb_charging_active = false;
    static bool solar_charging_active = false;
    static bool battery_low = false;
    static bool battery_full = false;

    if (charge_current_usb > PowerManager::USB_CURRENT_THRESHOLD && !usb_charging_active) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::USB_CONNECTED);
        usb_charging_active = true;
    }
    else if (charge_current_usb < PowerManager::USB_CURRENT_THRESHOLD && usb_charging_active) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::USB_DISCONNECTED);
        usb_charging_active = false;
    }

    if (charge_current_solar > PowerManager::SOLAR_CURRENT_THRESHOLD && !solar_charging_active) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::SOLAR_ACTIVE);
        solar_charging_active = true;
    }
    else if (charge_current_solar < PowerManager::SOLAR_CURRENT_THRESHOLD && solar_charging_active) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::SOLAR_INACTIVE);
        solar_charging_active = false;
    }

    if (battery_voltage < PowerManager::BATTERY_LOW_THRESHOLD && !battery_low) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_LOW);
```

```
        battery_low = true;
        battery_full = false; // Cancel overcharge event
    }
    else if (battery_voltage > PowerManager::BATTERY_FULL_THRESHOLD && !battery_full) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_FULL);
        battery_full = true;
        battery_low = false; // Cancel low battery event
    }
    else if (battery_voltage > PowerManager::BATTERY_LOW_THRESHOLD && battery_low) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_NORMAL);
        battery_low = false;
    }
    else if (battery_voltage < PowerManager::BATTERY_FULL_THRESHOLD && battery_full) {
        EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_NORMAL);
        battery_full = false;
    }
}

void TelemetryManager::collect_gps_telemetry(TelemetryRecord& record) {
    auto& nmea_data = NMEAData::get_instance();
    // Get GPS RMC data
    std::vector<std::string> rmc_tokens = nmea_data.get_rmc_tokens();
    if (rmc_tokens.size() >= 12) {  // RMC has at least 12 fields when complete
        record.time = rmc_tokens[1].substr(0, 6);  // Only keep HHMMSS
        record.latitude = rmc_tokens[3];
        record.lat_dir = rmc_tokens[4];
        record.longitude = rmc_tokens[5];
        record.lon_dir = rmc_tokens[6];
        record.speed = rmc_tokens[7];
        record.course = rmc_tokens[8];
        record.date = rmc_tokens[9];
    }
    else {
        // Fill with defaults if no GPS data
        record.time = "";
        record.latitude = "";
        record.lat_dir = "";
        record.longitude = "";
        record.lon_dir = "";
        record.speed = "";
        record.course = "";
        record.date = "";
    }

    // Get GPS GGA data
    std::vector<std::string> gga_tokens = nmea_data.get_gga_tokens();
    if (gga_tokens.size() >= 15) {  // GGA has 15 fields when complete
        record.fix_quality = gga_tokens[6];
        record.satellites = gga_tokens[7];
        record.altitude = gga_tokens[9];
    }
    else {
        // Fill with defaults if no GPS data
        record.fix_quality = "";
        record.satellites = "";
        record.altitude = "";
    }
}

void TelemetryManager::collect_sensor_telemetry(SensorDataRecord& sensor_record) {
    SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
    sensor_record.temperature = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
SensorDataTypeIdentifier::TEMPERATURE);
    sensor_record.pressure = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
SensorDataTypeIdentifier::PRESSURE);
    sensor_record.humidity = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
SensorDataTypeIdentifier::HUMIDITY);
    sensor_record.light = sensor_wrapper.sensor_read_data(SensorType::LIGHT,
SensorDataTypeIdentifier::LIGHT_LEVEL);
}

bool TelemetryManager::collect_telemetry() {
    uint32_t timestamp = DS3231::get_instance().get_local_time();
    TelemetryRecord record;
    record.timestamp = timestamp;
    record.build_version = std::to_string(BUILD_NUMBER);

    // Collect power telemetry and emit events
    collect_power_telemetry(record);
    emit_power_events(record.battery_voltage, record.charge_current_usb, record.charge_current_solar);

    // Collect GPS telemetry
    collect_gps_telemetry(record);

    // Collect sensor telemetry
    SensorDataRecord sensor_record;
    sensor_record.timestamp = timestamp;
```

```
    collect_sensor_telemetry(sensor_record);

    mutex_enter_blocking(&telemetry_mutex);

    telemetry_buffer[telemetry_buffer_write_index] = record;
    sensor_data_buffer[telemetry_buffer_write_index] = sensor_record;
    telemetry_buffer_write_index = (telemetry_buffer_write_index + 1) % TELEMETRY_BUFFER_SIZE;
    if (telemetry_buffer_count < TELEMETRY_BUFFER_SIZE) {
        telemetry_buffer_count++;
    }

    mutex_exit(&telemetry_mutex);

    uart_print("Telemetry collected", VerbosityLevel::DEBUG);

    return true;
}


bool TelemetryManager::flush_telemetry() {
    if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
        return false;
    }

    mutex_enter_blocking(&telemetry_mutex);

    if (telemetry_buffer_count == 0) {
        mutex_exit(&telemetry_mutex);
        return true; // Nothing to save
    }

    FILE* telemetry_file = fopen(TELEMETRY_CSV_PATH, "a");
    FILE* sensor_file = fopen(SENSOR_DATA_CSV_PATH, "a");

    if (!telemetry_file || !sensor_file) {
        uart_print("Failed to open telemetry or sensor log for writing", VerbosityLevel::ERROR);
        if (telemetry_file) fclose(telemetry_file);
        if (sensor_file) fclose(sensor_file);
        mutex_exit(&telemetry_mutex);
        return false;
    }

    // Calculate start index (for circular buffer)
    size_t read_index = 0;
    if (telemetry_buffer_count == TELEMETRY_BUFFER_SIZE) {
        // Buffer is full, start from oldest entry
        read_index = telemetry_buffer_write_index;
    }

    // Write all records to CSV
    for (size_t i = 0; i < telemetry_buffer_count; i++) {
        fprintf(telemetry_file, "%s\n", telemetry_buffer[read_index].to_csv().c_str());
        fprintf(sensor_file, "%s\n", sensor_data_buffer[read_index].to_csv().c_str());
        read_index = (read_index + 1) % TELEMETRY_BUFFER_SIZE;
    }

    // Clear buffer after successful write
    telemetry_buffer_count = 0;
    telemetry_buffer_write_index = 0;

    fclose(telemetry_file);
    fclose(sensor_file);

    mutex_exit(&telemetry_mutex);
    return true;
}

bool TelemetryManager::is_telemetry_collection_time(uint32_t current_time, uint32_t&
last_collection_time) {
    if (current_time - last_collection_time >= sample_interval_ms) {
        last_collection_time = current_time;
        return true;
    }
    return false;
}


bool TelemetryManager::is_telemetry_flush_time(uint32_t& collection_counter) {
    if (collection_counter >= flush_threshold) {
        collection_counter = 0;
        return true;
    }
    return false;
}

std::string TelemetryManager::get_last_telemetry_record_csv() {
    mutex_enter_blocking(&telemetry_mutex);
```

```
    if (telemetry_buffer_count == 0) {
        mutex_exit(&telemetry_mutex);
        return "";
    }

    TelemetryRecord last_record = get_last_telemetry_record();

    mutex_exit(&telemetry_mutex);

    return last_record.to_csv();
}

std::string TelemetryManager::get_last_sensor_record_csv() {
    mutex_enter_blocking(&telemetry_mutex);

    if (telemetry_buffer_count == 0) {
        mutex_exit(&telemetry_mutex);
        return "";
    }

    SensorDataRecord last_record = get_last_sensor_record();

    mutex_exit(&telemetry_mutex);

    return last_record.to_csv();
}
```

8.91 lib/telemetry/telemetry_manager.h File Reference

System telemetry collection and logging.

```
#include <cstdint>
#include <string>
#include "pico/stdlib.h"
#include "lib/location/NMEA/nmea_data.h"
#include "utils.h"
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include <functional>
```

Classes

- struct TelemetryRecord

  Structure representing a single telemetry data point.
- struct SensorDataRecord

  Structure representing a single sensor data point.
- class TelemetryManager

  Manages the collection, storage, and retrieval of telemetry data.

8.91.1 Detailed Description

System telemetry collection and logging.

This module handles periodic collection and storage of telemetry data from various satellite subsystems including power management, sensors (temperature, pressure, humidity, light), and GPS data.

Telemetry is collected at configurable intervals and stored in a circular buffer before being flushed to persistent storage after a configurable number of records are collected.

Definition in file telemetry_manager.h.

8.92 telemetry_manager.h

Go to the documentation of this file.

```



#ifndef TELEMETRY_MANAGER_H
#define TELEMETRY_MANAGER_H

#include <cstdint>
#include <string>
#include "pico/stdlib.h"
#include "lib/location/NMEA/nmea_data.h"
#include "utils.h"
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include <functional>

struct TelemetryRecord {
    uint32_t timestamp;

    std::string build_version;

    // Power data
    float battery_voltage;
    float system_voltage;
    float charge_current_usb;
    float charge_current_solar;
    float discharge_current;

    // GPS data - key RMC fields
    std::string time;
    std::string latitude;
    std::string lat_dir;
    std::string longitude;
    std::string lon_dir;
    std::string speed;
    std::string course;
    std::string date;

    // GPS data - key GGA fields
    std::string fix_quality;
    std::string satellites;
    std::string altitude;


    std::string to_csv() const {
        std::stringstream ss;
        ss « timestamp « ","
           « build_version « ","
           « std::fixed « std::setprecision(3)
           « battery_voltage « ","
           « system_voltage « ","
           « charge_current_usb « ","
           « charge_current_solar « ","
```

```
            « discharge_current « ","

            // GPS RMC data
            « time « ","
            « latitude « "," « lat_dir « ","
            « longitude « "," « lon_dir « ","
            « speed « ","
            « course « ","
            « date « ","
            // GPS GGA data
            « fix_quality « ","
            « satellites « ","
            « altitude;
        return ss.str();
    }
};


struct SensorDataRecord {
    uint32_t timestamp;
    float temperature;
    float pressure;
    float humidity;
    float light;

    std::string to_csv() const {
        std::stringstream ss;
        ss « timestamp « ","
           « std::fixed « std::setprecision(3)
           « temperature « ","
           « pressure « ","
           « humidity « ","
           « light;
        return ss.str();
    }
};


class TelemetryManager {
public:
    static TelemetryManager& get_instance() {
        static TelemetryManager instance;
        return instance;
    }

    bool init();

    bool collect_telemetry();

    void collect_power_telemetry(TelemetryRecord& record);

    void emit_power_events(float battery_voltage, float charge_current_usb, float charge_current_solar);

    void collect_gps_telemetry(TelemetryRecord& record);

    void collect_sensor_telemetry(SensorDataRecord& sensor_record);

    bool flush_telemetry();

    bool flush_sensor_data();

    bool is_telemetry_collection_time(uint32_t current_time, uint32_t& last_collection_time);

    bool is_telemetry_flush_time(uint32_t& collection_counter);


    std::string get_last_telemetry_record_csv();

    std::string get_last_sensor_record_csv();

    static constexpr int TELEMETRY_BUFFER_SIZE = 20;

    TelemetryRecord& get_last_telemetry_record() {
        size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) % TELEMETRY_BUFFER_SIZE;
        return telemetry_buffer[last_record_index];
    }

    SensorDataRecord& get_last_sensor_record() {
        size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) % TELEMETRY_BUFFER_SIZE;
        return sensor_data_buffer[last_record_index];
    }

    size_t get_telemetry_buffer_count() const { return telemetry_buffer_count; }
    size_t get_telemetry_buffer_write_index() const { return telemetry_buffer_write_index; }
```

```

private:
    TelemetryManager();  // Private constructor
    ~TelemetryManager() = default;

    static constexpr uint32_t DEFAULT_SAMPLE_INTERVAL_MS = 1000;

    static constexpr uint32_t DEFAULT_FLUSH_THRESHOLD = 10;

    uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS;

    uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD;

    TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE];
    size_t telemetry_buffer_count = 0;
    size_t telemetry_buffer_write_index = 0;

    SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE];

    mutex_t telemetry_mutex;
};
#endif // TELEMETRY_MANAGER_H
 // End of TelemetryManager group
```

8.93 lib/utils.cpp File Reference

Implementation of utility functions for the Kubisat firmware.

```
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
#include <vector>
#include <queue>
#include <string>
#include <array>
#include "system_state_manager.h"
```

Functions

- std::string get_level_color (VerbosityLevel level)

    Gets ANSI color code for verbosity level.
- std::string get_level_prefix (VerbosityLevel level)

    Gets text prefix for verbosity level.
- void uart_print (const std::string &msg, VerbosityLevel level, uart_inst_t *uart)

    Prints a message to the UART with a timestamp and core number.

Variables

- static mutex_t uart_mutex

    Mutex for UART access protection.

8.93.1 Detailed Description

Implementation of utility functions for the Kubisat firmware.

Definition in file utils.cpp.

8.93.2 Function Documentation

8.93.2.1 get_level_color()

```
std::string get_level_color (
            VerbosityLevel level)
```

Gets ANSI color code for verbosity level.

Parameters

level	The verbosity level

Returns

ANSI color escape sequence

Definition at line 25 of file utils.cpp.

8.93.2.2 get_level_prefix()

```
std::string get_level_prefix (
            VerbosityLevel level)
```

Gets text prefix for verbosity level.

Parameters

level	The verbosity level

Returns

Text prefix for the level

Definition at line 41 of file utils.cpp.

8.93.2.3 uart_print()

```
void uart_print (
            const std::string & msg,
            VerbosityLevel level,
            uart_inst_t * uart)
```

Prints a message to the UART with a timestamp and core number.

Prints a message to UART with timestamp and formatting.

Parameters

msg	The message to print.
uart	The UART instance to use for printing.

Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 58 of file utils.cpp.

8.93.3 Variable Documentation

8.93.3.1 uart_mutex

```
mutex_t uart_mutex  [static]
```

Mutex for UART access protection.

Definition at line 17 of file utils.cpp.

8.94 utils.cpp

Go to the documentation of this file.

```
00001 #include "utils.h"
00002 #include "pico/multicore.h"
00003 #include "pico/sync.h"
00004 #include <vector>
00005 #include <queue>
00006 #include <string>
00007 #include <array>
00008 #include "system_state_manager.h"
00009
00014
00015
00017 static mutex_t uart_mutex;
00018
00019
00025 std::string get_level_color(VerbosityLevel level) {
00026     switch (level) {
00027         case VerbosityLevel::ERROR:   return ANSI_RED;
00028         case VerbosityLevel::WARNING: return ANSI_YELLOW;
00029         case VerbosityLevel::INFO:    return ANSI_GREEN;
00030         case VerbosityLevel::DEBUG:   return ANSI_BLUE;
00031         default:                      return "";
00032     }
00033 }
00034
00035
00041 std::string get_level_prefix(VerbosityLevel level) {
00042     switch (level) {
00043         case VerbosityLevel::ERROR:   return "ERROR: ";
00044         case VerbosityLevel::WARNING: return "WARNING: ";
00045         case VerbosityLevel::INFO:    return "INFO: ";
00046         case VerbosityLevel::DEBUG:   return "DEBUG: ";
00047         default:                      return "";
00048     }
00049 }
00050
00058 void uart_print(const std::string& msg, VerbosityLevel level, uart_inst_t* uart) {
00059     if (static_cast<int>(level) >
      static_cast<int>(SystemStateManager::get_instance().get_uart_verbosity())) {
00060         return;
00061     }
00062
00063     static bool mutex_inited = false;
00064     if (!mutex_inited) {
```

```
        mutex_init(&uart_mutex);
        mutex_inited = true;
    }

    uint32_t timestamp = to_ms_since_boot(get_absolute_time());
    uint core_num = get_core_num();

    std::string color = get_level_color(level);
    std::string prefix = get_level_prefix(level);
    std::string msg_to_send = "[" + std::to_string(timestamp) + "ms] - Core " +
                              std::to_string(core_num) + ": " +
                              color + prefix + ANSI_RESET + msg + "\r\n";

    mutex_enter_blocking(&uart_mutex);
    uart_puts(uart, msg_to_send.c_str());
    mutex_exit(&uart_mutex);
}
```

8.95 lib/utils.h File Reference

Utility functions and definitions for the Kubisat firmware.

```
#include <stdio.h>
#include <string>
#include "pico/stdlib.h"
#include "hardware/uart.h"
#include "pin_config.h"
#include <vector>
```

Macros

- #define ANSI_RED "\033[31m"

  ANSI escape codes for terminal color output.
- #define ANSI_GREEN "\033[32m"
- #define ANSI_YELLOW "\033[33m"
- #define ANSI_BLUE "\033[34m"
- #define ANSI_RESET "\033[0m"

Enumerations

- enum class VerbosityLevel {
  SILENT = 0 , ERROR = 1 , WARNING = 2 , INFO = 3 ,
  DEBUG = 4 }

  Verbosity levels for logging system.

Functions

- void uart_print (const std::string &msg, VerbosityLevel level=VerbosityLevel::INFO, uart_inst_t ∗uart=DEBUG_UART_PORT)

  Prints a message to UART with timestamp and formatting.

8.95.1 Detailed Description

Utility functions and definitions for the Kubisat firmware.

Contains UART logging, color definitions, and CRC calculations

Definition in file utils.h.

8.95.2 Macro Definition Documentation

8.95.2.1 ANSI_RED

```
#define ANSI_RED "\033[31m"
```

ANSI escape codes for terminal color output.

Definition at line 20 of file utils.h.

8.95.2.2 ANSI_GREEN

```
#define ANSI_GREEN "\033[32m"
```

Definition at line 21 of file utils.h.

8.95.2.3 ANSI_YELLOW

```
#define ANSI_YELLOW "\033[33m"
```

Definition at line 22 of file utils.h.

8.95.2.4 ANSI_BLUE

```
#define ANSI_BLUE "\033[34m"
```

Definition at line 23 of file utils.h.

8.95.2.5 ANSI_RESET

```
#define ANSI_RESET "\033[0m"
```

Definition at line 24 of file utils.h.

8.95.3 Enumeration Type Documentation

8.95.3.1 VerbosityLevel

```
enum class VerbosityLevel  [strong]
```

Verbosity levels for logging system.

Enumerator

SILENT	No output
ERROR	Only critical errors
WARNING	Warnings and errors
INFO	Normal operation information
DEBUG	Detailed debug information

Definition at line 30 of file utils.h.

8.95.4 Function Documentation

8.95.4.1 uart_print()

```
void uart_print (
              const std::string & msg,
              VerbosityLevel level,
              uart_inst_t * uart)
```

Prints a message to UART with timestamp and formatting.

Parameters

msg	The message to print
level	Message verbosity level
uart	The UART port to use

Prints a message to UART with timestamp and formatting.

Parameters

msg	The message to print.
uart	The UART instance to use for printing.

Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 58 of file utils.cpp.

8.96 utils.h

Go to the documentation of this file.

```
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
#include <string>
#include "pico/stdlib.h"
#include "hardware/uart.h"
#include "pin_config.h"
#include <vector>




#define ANSI_RED     "\033[31m"
#define ANSI_GREEN   "\033[32m"
#define ANSI_YELLOW  "\033[33m"
#define ANSI_BLUE    "\033[34m"
#define ANSI_RESET   "\033[0m"


enum class VerbosityLevel {
    SILENT = 0,
    ERROR = 1,
    WARNING = 2,
    INFO = 3,
    DEBUG = 4
};


void uart_print(const std::string& msg,
                VerbosityLevel level = VerbosityLevel::INFO,
                uart_inst_t* uart = DEBUG_UART_PORT);


#endif
```

8.97 main.cpp File Reference

```
#include "includes.h"
```

Macros

- #define LOG_FILENAME "/log.txt"

Functions

- void core1_entry ()
- bool init_pico_hw ()
- bool init_modules ()
- int main ()

Variables

- char buffer [BUFFER_SIZE] = {0}
- int buffer_index = 0

8.97.1 Macro Definition Documentation

8.97.1.1 LOG_FILENAME

```
#define LOG_FILENAME "/log.txt"
```

Definition at line 3 of file main.cpp.

8.97.2 Function Documentation

8.97.2.1 core1_entry()

```
void core1_entry ()
```

Definition at line 8 of file main.cpp.

8.97.2.2 init_pico_hw()

```
bool init_pico_hw ()
```

Definition at line 53 of file main.cpp.

8.97.2.3 init_modules()

```
bool init_modules ()
```

Definition at line 99 of file main.cpp.

8.97.2.4 main()

```
int main (
            void )
```

Definition at line 160 of file main.cpp.

8.97.3 Variable Documentation

8.97.3.1 buffer

```
char buffer[BUFFER_SIZE] = {0}
```

Definition at line 5 of file main.cpp.

8.97.3.2 buffer_index

```
int buffer_index = 0
```

Definition at line 6 of file main.cpp.

8.98 main.cpp

Go to the documentation of this file.

```
#include "includes.h"

#define LOG_FILENAME "/log.txt"

char buffer[BUFFER_SIZE] = {0};
int buffer_index = 0;

void core1_entry() {
    uart_print("Starting core 1", VerbosityLevel::DEBUG);
    EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::CORE1_START);

    uint32_t last_clock_check_time = 0;
    uint32_t last_telemetry_time = 0;
    uint32_t telemetry_collection_counter = 0;

    TelemetryManager::get_instance().init();

    while (true) {
        collect_gps_data();

        uint32_t currentTime = to_ms_since_boot(get_absolute_time());

        uint32_t check_interval_ms = DS3231::get_instance().get_clock_sync_interval() * 60000;
        if (currentTime - last_clock_check_time >= check_interval_ms) {
            last_clock_check_time = currentTime;

            if (DS3231::get_instance().is_sync_needed()) {
                uart_print("Clock sync interval reached, attempting sync", VerbosityLevel::INFO);
                DS3231::get_instance().sync_clock_with_gps();
```

```
            }
        }

        if (TelemetryManager::get_instance().is_telemetry_collection_time(currentTime, last_telemetry_time)) {
            TelemetryManager::get_instance().collect_telemetry();
            telemetry_collection_counter++;

            if (TelemetryManager::get_instance().is_telemetry_flush_time(telemetry_collection_counter)) {
                TelemetryManager::get_instance().flush_telemetry();
                telemetry_collection_counter = 0;
            }
        }

        if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
            sleep_ms(100);
            uart_print("Entering BOOTSEL mode...", VerbosityLevel::WARNING);
            reset_usb_boot(0, 0);
        }

        sleep_ms(10);
    }
}

bool init_pico_hw() {
    stdio_init_all();

    uart_init(DEBUG_UART_PORT, DEBUG_UART_BAUD_RATE);
    gpio_set_function(DEBUG_UART_TX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_TX_PIN));
    gpio_set_function(DEBUG_UART_RX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_RX_PIN));

    uart_init(GPS_UART_PORT, GPS_UART_BAUD_RATE);
    gpio_set_function(GPS_UART_TX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_TX_PIN));
    gpio_set_function(GPS_UART_RX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_RX_PIN));

    gpio_init(PICO_DEFAULT_LED_PIN);
    gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);

    gpio_init(PICO_DEFAULT_LED_PIN);
    gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
    gpio_put(PICO_DEFAULT_LED_PIN, 1);

    i2c_init(MAIN_I2C_PORT, 400 * 1000);
    gpio_set_function(MAIN_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_set_function(MAIN_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(MAIN_I2C_SCL_PIN);
    gpio_pull_up(MAIN_I2C_SDA_PIN);

    gpio_init(GPS_POWER_ENABLE_PIN);
    gpio_set_dir(GPS_POWER_ENABLE_PIN, GPIO_OUT);
    gpio_put(GPS_POWER_ENABLE_PIN, 1);

    i2c_init(SENSORS_I2C_PORT, 400 * 1000);
    gpio_set_function(SENSORS_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_set_function(SENSORS_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(SENSORS_I2C_SCL_PIN);
    gpio_pull_up(SENSORS_I2C_SDA_PIN);
    gpio_init(SENSORS_POWER_ENABLE_PIN);
    gpio_set_dir(SENSORS_POWER_ENABLE_PIN, GPIO_OUT);
    gpio_put(SENSORS_POWER_ENABLE_PIN, 1);

    SystemStateManager::get_instance();

    EventEmitter::emit(EventGroup::GPS, GPSEvent::POWER_ON);

    system("color");

    return true;
}

bool init_modules(){
    bool radio_init_status = initialize_radio();
    SystemStateManager::get_instance().set_radio_init_ok(radio_init_status);

    bool sd_init_status = fs_init();
    SystemStateManager::get_instance().set_sd_card_mounted(sd_init_status);

    if (sd_init_status) {
        FILE *fp = fopen(LOG_FILENAME, "w");
        if (fp) {
            uart_print("Log file opened.", VerbosityLevel::DEBUG);
            int bytes_written = fprintf(fp, "System init started.\n");
            uart_print("Written " + std::to_string(bytes_written) + " bytes.", VerbosityLevel::DEBUG);
            int close_status = fclose(fp);
            uart_print("Close file status: " + std::to_string(close_status), VerbosityLevel::DEBUG);

```

```
            struct stat file_stat;
            if (stat(LOG_FILENAME, &file_stat) == 0) {
                size_t file_size = file_stat.st_size;
                uart_print("File size: " + std::to_string(file_size) + " bytes", VerbosityLevel::DEBUG);
            } else {
                uart_print("Failed to get file size", VerbosityLevel::ERROR);
            }

            uart_print("File path: /" + std::string(LOG_FILENAME), VerbosityLevel::DEBUG);
        } else {
            uart_print("Failed to open log file for writing.", VerbosityLevel::ERROR);
        }
    }

    if (sd_init_status) {
        uart_print("SD card init: OK", VerbosityLevel::DEBUG);
    } else {
        uart_print("SD card init: FAILED", VerbosityLevel::ERROR);
    }

    if (radio_init_status) {
        uart_print("Radio init: OK", VerbosityLevel::DEBUG);
    } else {
        uart_print("Radio init: FAILED", VerbosityLevel::ERROR);
    }

    Frame boot = frame_build(OperationType::RES, 0, 0, "HELLO");
    send_frame_lora(boot);

    uart_print("Initializing sensors...", VerbosityLevel::DEBUG);

    SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
    bool light_sensor_init = sensor_wrapper.sensor_init(SensorType::LIGHT, SENSORS_I2C_PORT);
    SystemStateManager::get_instance().set_light_sensor_init_ok(light_sensor_init);

    bool env_sensor_init = sensor_wrapper.sensor_init(SensorType::ENVIRONMENT, SENSORS_I2C_PORT);
    SystemStateManager::get_instance().set_env_sensor_init_ok(env_sensor_init);

    if (!light_sensor_init || !env_sensor_init) {
        uart_print("One or more sensors failed to initialize", VerbosityLevel::WARNING);
    }

    return sd_init_status && radio_init_status;
}

int main()
{
    init_pico_hw();
    sleep_ms(100);
    init_modules();
    EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::BOOT);
    sleep_ms(100);
    multicore_launch_core1(core1_entry);

    gpio_put(PICO_DEFAULT_LED_PIN, 0);

    bool power_manager_init_status = PowerManager::get_instance().initialize();
    if (power_manager_init_status) {
        std::map<std::string, std::string> power_config = {
            {"operating_mode", "continuous"},
            {"averaging_mode", "16"},
        };
        PowerManager::get_instance().configure(power_config);
    } else {
        uart_print("Power manager init error", VerbosityLevel::ERROR);
    }

    Frame boot = frame_build(OperationType::RES, 0, 0, "START");
    send_frame_lora(boot);

    std::string boot_string = "System init completed @ " + std::to_string(to_ms_since_boot(get_absolute_time())) + " ms";
    uart_print(boot_string, VerbosityLevel::WARNING);

    gpio_put(PICO_DEFAULT_LED_PIN, 1);

    while (true)
    {
        int packet_size = LoRa.parse_packet();
        if (packet_size)
        {
            on_receive(packet_size);
        }

        handle_uart_input();
    }
```

```

    return 0;
}
```

8.99 test/comms/commands/test_clock_commands.cpp File Reference

8.100 test_clock_commands.cpp

Go to the documentation of this file.

8.101 test/comms/commands/test_diagnostic_commands.cpp File Reference

```
#include "unity.h"
#include "commands.h"
#include "protocol.h"
#include "build_number.h"
```

Functions

- void test_handle_get_commands_list (void)
- void test_handle_get_build_version (void)
- void test_handle_verbosity (void)
- void test_handle_enter_bootloader_mode (void)

8.101.1 Function Documentation

8.101.1.1 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
            void )
```

Definition at line 6 of file test_diagnostic_commands.cpp.

8.101.1.2 test_handle_get_build_version()

```
void test_handle_get_build_version (
            void )
```

Definition at line 15 of file test_diagnostic_commands.cpp.

8.101.1.3 test_handle_verbosity()

```
void test_handle_verbosity (
            void )
```

Definition at line 25 of file test_diagnostic_commands.cpp.

8.101.1.4 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
            void )
```

Definition at line 40 of file test_diagnostic_commands.cpp.

8.102 test_diagnostic_commands.cpp

Go to the documentation of this file.

```
00001 #include "unity.h"
00002 #include "commands.h"
00003 #include "protocol.h"
00004 #include "build_number.h"
00005
00006 void test_handle_get_commands_list(void) {
00007     std::vector<Frame> response = handle_get_commands_list("", OperationType::GET);
00008
00009     TEST_ASSERT_TRUE(response.size() > 0);
00010     TEST_ASSERT_EQUAL(OperationType::SEQ, response[0].operationType);
00011     TEST_ASSERT_EQUAL(1, response[0].group);
00012     TEST_ASSERT_EQUAL(0, response[0].command);
00013 }
00014
00015 void test_handle_get_build_version(void) {
00016     std::vector<Frame> response = handle_get_build_version("", OperationType::GET);
00017
00018     TEST_ASSERT_EQUAL(1, response.size());
00019     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00020     TEST_ASSERT_EQUAL(1, response[0].group);
00021     TEST_ASSERT_EQUAL(1, response[0].command);
00022     TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
00023 }
00024
00025 void test_handle_verbosity(void) {
00026     // Test SET operation
00027     std::vector<Frame> response = handle_verbosity("2", OperationType::SET);
00028     TEST_ASSERT_EQUAL(1, response.size());
00029     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00030     TEST_ASSERT_EQUAL(1, response[0].group);
00031     TEST_ASSERT_EQUAL(8, response[0].command);
00032     TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());
00033
00034     // Test GET operation
00035     response = handle_verbosity("", OperationType::GET);
00036     TEST_ASSERT_EQUAL(1, response.size());
00037     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00038 }
00039
00040 void test_handle_enter_bootloader_mode(void) {
00041     std::vector<Frame> response = handle_enter_bootloader_mode("", OperationType::SET);
00042
00043     TEST_ASSERT_EQUAL(1, response.size());
00044     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00045     TEST_ASSERT_EQUAL(1, response[0].group);
00046     TEST_ASSERT_EQUAL(9, response[0].command);
00047 }
```

8.103 test/comms/commands/test_event_commands.cpp File Reference

8.104 test_event_commands.cpp

Go to the documentation of this file.

```
00001
```

8.105 test/comms/commands/test_gps_commands.cpp File Reference

8.106 test_gps_commands.cpp

Go to the documentation of this file.

```
00001
```

8.107 test/comms/commands/test_power_commands.cpp File Reference

8.108 test_power_commands.cpp

Go to the documentation of this file.

```
00001
```

8.109 test/comms/commands/test_sensor_commands.cpp File Reference

8.110 test_sensor_commands.cpp

Go to the documentation of this file.

```
00001
```

8.111 test/comms/commands/test_storage_commands.cpp File Reference

8.112 test_storage_commands.cpp

Go to the documentation of this file.

```
00001
```

8.113 test/comms/commands/test_telemetry_commands.cpp File Reference

8.114 test_telemetry_commands.cpp

Go to the documentation of this file.

```
00001
```

8.115 test/comms/test_comand_handlers.cpp File Reference

```
#include "unity.h"
#include "protocol.h"
#include "communication.h"
#include "commands.h"
```

Functions

- void send_frame_uart (const Frame &frame)
- void send_frame_lora (const Frame &frame)
- void setUp (void)
- void tearDown (void)
- void test_command_handler_get_operation (void)
- void test_command_handler_set_operation (void)
- void test_command_handler_invalid_operation (void)

Variables

- static bool uart_send_called = false
- static bool lora_send_called = false
- static Frame last_frame_sent

8.115.1 Function Documentation

8.115.1.1 send_frame_uart()

```
void send_frame_uart (
            const Frame & frame)
```

Definition at line 11 of file test_comand_handlers.cpp.

8.115.1.2 send_frame_lora()

```
void send_frame_lora (
            const Frame & frame)
```

Definition at line 16 of file test_comand_handlers.cpp.

8.115.1.3 setUp()

```
void setUp (
            void )
```

Definition at line 21 of file test_comand_handlers.cpp.

8.115.1.4 tearDown()

```
void tearDown (
            void )
```

Definition at line 26 of file test_comand_handlers.cpp.

8.115.1.5 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
            void )
```

Definition at line 29 of file test_comand_handlers.cpp.

8.115.1.6 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
            void )
```

Definition at line 39 of file test_comand_handlers.cpp.

8.115.1.7 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
            void )
```

Definition at line 54 of file test_comand_handlers.cpp.

8.115.2 Variable Documentation

8.115.2.1 uart_send_called

```
bool uart_send_called = false  [static]
```

Definition at line 7 of file test_comand_handlers.cpp.

8.115.2.2 lora_send_called

```
bool lora_send_called = false  [static]
```

Definition at line 8 of file test_comand_handlers.cpp.

8.115.2.3 last_frame_sent

```
Frame last_frame_sent [static]
```

Definition at line 9 of file test_comand_handlers.cpp.

8.116 test_comand_handlers.cpp

Go to the documentation of this file.

```
00001 // test/comms/test_command_handlers.cpp
00002 #include "unity.h"
00003 #include "protocol.h"
00004 #include "communication.h"
00005 #include "commands.h"
00006
00007 static bool uart_send_called = false;
00008 static bool lora_send_called = false;
00009 static Frame last_frame_sent;
00010
00011 void send_frame_uart(const Frame& frame) {
00012     uart_send_called = true;
00013     last_frame_sent = frame;
00014 }
00015
00016 void send_frame_lora(const Frame& frame) {
00017     lora_send_called = true;
00018     last_frame_sent = frame;
00019 }
00020
00021 void setUp(void) {
00022     uart_send_called = false;
00023     lora_send_called = false;
00024 }
00025
00026 void tearDown(void) {
00027 }
00028
00029 void test_command_handler_get_operation(void) {
00030     std::vector<Frame> response = handle_get_build_version("", OperationType::GET);
00031
00032     TEST_ASSERT_EQUAL(1, response.size());
00033     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00034     TEST_ASSERT_EQUAL(1, response[0].group);
00035     TEST_ASSERT_EQUAL(1, response[0].command);
00036     TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
00037 }
00038
00039 void test_command_handler_set_operation(void) {
00040     VerbosityLevel old_level = get_verbosity_level();
00041     std::vector<Frame> response = handle_verbosity("2", OperationType::SET);
00042
00043     TEST_ASSERT_EQUAL(1, response.size());
00044     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00045     TEST_ASSERT_EQUAL(1, response[0].group);
00046     TEST_ASSERT_EQUAL(8, response[0].command);
00047     TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());
00048
00049     TEST_ASSERT_EQUAL(VerbosityLevel::WARNING, get_verbosity_level());
00050
00051     set_verbosity_level(old_level);
00052 }
00053
00054 void test_command_handler_invalid_operation(void) {
00055     std::vector<Frame> response = handle_get_build_version("", OperationType::SET);
00056
00057     TEST_ASSERT_EQUAL(1, response.size());
00058     TEST_ASSERT_EQUAL(OperationType::ERR, response[0].operationType);
00059     TEST_ASSERT_EQUAL(1, response[0].group);
00060     TEST_ASSERT_EQUAL(1, response[0].command);
00061 }
```

8.117 test/comms/test_converters.cpp File Reference

```
#include "test_frame_common.h"
```

Functions

- void test_operation_type_conversion ()
- void test_value_unit_type_conversion ()
- void test_exception_type_conversion ()
- void test_hex_string_conversion ()

8.117.1 Function Documentation

8.117.1.1 test_operation_type_conversion()

```
void test_operation_type_conversion (
            void )
```

Definition at line 4 of file test_converters.cpp.

8.117.1.2 test_value_unit_type_conversion()

```
void test_value_unit_type_conversion (
            void )
```

Definition at line 13 of file test_converters.cpp.

8.117.1.3 test_exception_type_conversion()

```
void test_exception_type_conversion (
            void )
```

Definition at line 20 of file test_converters.cpp.

8.117.1.4 test_hex_string_conversion()

```
void test_hex_string_conversion (
            void )
```

Definition at line 27 of file test_converters.cpp.

8.118 test_converters.cpp

Go to the documentation of this file.

```
// test_frame_converters.cpp
#include "test_frame_common.h"

void test_operation_type_conversion() {
    OperationType type = OperationType::GET;
    std::string str = operation_type_to_string(type);
    OperationType converted = string_to_operation_type(str);

    TEST_ASSERT_EQUAL(type, converted);
    TEST_ASSERT_EQUAL_STRING("GET", str.c_str());
}

void test_value_unit_type_conversion() {
    ValueUnit unit = ValueUnit::VOLT;
    std::string str = value_unit_type_to_string(unit);

    TEST_ASSERT_EQUAL_STRING("V", str.c_str());
}

void test_exception_type_conversion() {
    ExceptionType type = ExceptionType::INVALID_PARAM;
    std::string str = exception_type_to_string(type);

    TEST_ASSERT_EQUAL_STRING("INVALID PARAM", str.c_str());
}

void test_hex_string_conversion() {
    std::string hex = "0A0B0C";
    std::vector<uint8_t> bytes = hex_string_to_bytes(hex);

    TEST_ASSERT_EQUAL(3, bytes.size());
    TEST_ASSERT_EQUAL(0x0A, bytes[0]);
    TEST_ASSERT_EQUAL(0x0B, bytes[1]);
    TEST_ASSERT_EQUAL(0x0C, bytes[2]);
}
```

8.119 test/comms/test_frame_build.cpp File Reference

```
#include "test_frame_common.h"
```

Functions

- void test_frame_build_val ()
- void test_frame_build_err ()
- void test_frame_build_get ()
- void test_frame_build_set ()
- void test_frame_build_res ()
- void test_frame_build_seq ()

8.119.1 Function Documentation

8.119.1.1 test_frame_build_val()

```
void test_frame_build_val (
            void )
```

Definition at line 4 of file test_frame_build.cpp.

8.119.1.2 test_frame_build_err()

```
void test_frame_build_err (
            void )
```

Definition at line 15 of file test_frame_build.cpp.

8.119.1.3 test_frame_build_get()

```
void test_frame_build_get (
            void )
```

Definition at line 24 of file test_frame_build.cpp.

8.119.1.4 test_frame_build_set()

```
void test_frame_build_set (
            void )
```

Definition at line 35 of file test_frame_build.cpp.

8.119.1.5 test_frame_build_res()

```
void test_frame_build_res (
            void )
```

Definition at line 46 of file test_frame_build.cpp.

8.119.1.6 test_frame_build_seq()

```
void test_frame_build_seq (
            void )
```

Definition at line 57 of file test_frame_build.cpp.

8.120 test_frame_build.cpp

Go to the documentation of this file.

```
// test_frame_build.cpp
#include "test_frame_common.h"

void test_frame_build_val() {
    Frame frame = frame_build(OperationType::VAL, 1, 2, "test_value", ValueUnit::VOLT);

    TEST_ASSERT_EQUAL(1, frame.direction);
    TEST_ASSERT_EQUAL(OperationType::VAL, frame.operationType);
    TEST_ASSERT_EQUAL(1, frame.group);
    TEST_ASSERT_EQUAL(2, frame.command);
    TEST_ASSERT_EQUAL_STRING("test_value", frame.value.c_str());
    TEST_ASSERT_EQUAL_STRING("V", frame.unit.c_str());
}

void test_frame_build_err() {
    Frame frame = frame_build(OperationType::ERR, 1, 2, "error_message");

    TEST_ASSERT_EQUAL(1, frame.direction);
    TEST_ASSERT_EQUAL(OperationType::ERR, frame.operationType);
    TEST_ASSERT_EQUAL_STRING("error_message", frame.value.c_str());
    TEST_ASSERT_EQUAL_STRING("", frame.unit.c_str());
}

void test_frame_build_get() {
    Frame frame = frame_build(OperationType::GET, 3, 4, "");

    TEST_ASSERT_EQUAL(0, frame.direction);  // Ground to satellite
    TEST_ASSERT_EQUAL(OperationType::GET, frame.operationType);
    TEST_ASSERT_EQUAL(3, frame.group);
    TEST_ASSERT_EQUAL(4, frame.command);
    TEST_ASSERT_EQUAL_STRING("", frame.value.c_str());
    TEST_ASSERT_EQUAL_STRING("", frame.unit.c_str());
}

void test_frame_build_set() {
    Frame frame = frame_build(OperationType::SET, 5, 6, "set_value", ValueUnit::SECOND);

    TEST_ASSERT_EQUAL(0, frame.direction);  // Ground to satellite
    TEST_ASSERT_EQUAL(OperationType::SET, frame.operationType);
    TEST_ASSERT_EQUAL(5, frame.group);
    TEST_ASSERT_EQUAL(6, frame.command);
    TEST_ASSERT_EQUAL_STRING("set_value", frame.value.c_str());
    TEST_ASSERT_EQUAL_STRING("s", frame.unit.c_str());  // Assuming SECOND is converted to "s"
}

void test_frame_build_res() {
    Frame frame = frame_build(OperationType::RES, 7, 8, "result", ValueUnit::BOOL);

    TEST_ASSERT_EQUAL(1, frame.direction);  // Satellite to ground
    TEST_ASSERT_EQUAL(OperationType::RES, frame.operationType);
    TEST_ASSERT_EQUAL(7, frame.group);
    TEST_ASSERT_EQUAL(8, frame.command);
    TEST_ASSERT_EQUAL_STRING("result", frame.value.c_str());
    TEST_ASSERT_EQUAL_STRING("bool", frame.unit.c_str());  // Assuming BOOL is converted to "bool"
}

void test_frame_build_seq() {
    Frame frame = frame_build(OperationType::SEQ, 9, 10, "sequence_data", ValueUnit::TEXT);

    TEST_ASSERT_EQUAL(1, frame.direction);  // Satellite to ground
    TEST_ASSERT_EQUAL(OperationType::SEQ, frame.operationType);
    TEST_ASSERT_EQUAL(9, frame.group);
    TEST_ASSERT_EQUAL(10, frame.command);
    TEST_ASSERT_EQUAL_STRING("sequence_data", frame.value.c_str());
    TEST_ASSERT_EQUAL_STRING("text", frame.unit.c_str());  // Assuming TEXT is converted to "text"
}
```

8.121 test/comms/test_frame_coding.cpp File Reference

```
#include "test_frame_common.h"
```

Functions

- void test_frame_encode_basic ()
- void test_frame_decode_basic ()
- void test_frame_decode_invalid_header ()

8.121.1 Function Documentation

8.121.1.1 test_frame_encode_basic()

```
void test_frame_encode_basic (
            void )
```

Definition at line 4 of file test_frame_coding.cpp.

8.121.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
            void )
```

Definition at line 16 of file test_frame_coding.cpp.

8.121.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
            void )
```

Definition at line 29 of file test_frame_coding.cpp.

8.122 test_frame_coding.cpp

Go to the documentation of this file.

```
00001 // test_frame_codec.cpp
00002 #include "test_frame_common.h"
00003
00004 void test_frame_encode_basic() {
00005     uart_puts(uart0, "start frame_encode_basic");
00006     Frame frame = create_test_frame();
00007     std::string encoded = frame_encode(frame);
00008
00009     TEST_ASSERT_NOT_EQUAL(0, encoded.length());
00010     TEST_ASSERT_TRUE(encoded.find(FRAME_BEGIN) != std::string::npos);
00011     TEST_ASSERT_TRUE(encoded.find(FRAME_END) != std::string::npos);
00012     TEST_ASSERT_TRUE(encoded.find("test_value") != std::string::npos);
00013     uart_puts(uart0, "stop frame_encode_basic");
00014 }
00015
00016 void test_frame_decode_basic() {
00017
00018     Frame original = create_test_frame();
00019     std::string encoded = frame_encode(original);
00020     Frame decoded = frame_decode(encoded);
00021
00022     TEST_ASSERT_EQUAL(original.direction, decoded.direction);
00023     TEST_ASSERT_EQUAL(original.group, decoded.group);
00024     TEST_ASSERT_EQUAL(original.command, decoded.command);
00025     TEST_ASSERT_EQUAL_STRING(original.value.c_str(), decoded.value.c_str());
00026     TEST_ASSERT_EQUAL_STRING(original.unit.c_str(), decoded.unit.c_str());
```

```
}

void test_frame_decode_invalid_header() {
    std::string invalid_frame = "INVALID" + std::string(1, DELIMITER) + "rest_of_frame";
    bool exceptionThrown = false;

    try {
        Frame decoded = frame_decode(invalid_frame);
    } catch (const std::runtime_error& e) {
        exceptionThrown = true;
    } catch (...) {
        // Catch any other exceptions to avoid crashing the test
    }

    TEST_ASSERT_TRUE(exceptionThrown);
}
```

8.123 test/comms/test_frame_common.h File Reference

```
#include "unity.h"
#include "protocol.h"
#include "communication.h"
```

Functions

- Frame create_test_frame ()

8.123.1 Function Documentation

8.123.1.1 create_test_frame()

```
Frame create_test_frame ()   [inline]
```

Definition at line 9 of file test_frame_common.h.

8.124 test_frame_common.h

Go to the documentation of this file.

```
00001 // test_frame_common.h
00002 #ifndef TEST_FRAME_COMMON_H
00003 #define TEST_FRAME_COMMON_H
00004
00005 #include "unity.h"
00006 #include "protocol.h"
00007 #include "communication.h"
00008
00009 inline Frame create_test_frame() {
00010     Frame frame;
00011     frame.header = FRAME_BEGIN;
00012     frame.direction = 1;
00013     frame.operationType = OperationType::GET;
00014     frame.group = 1;
00015     frame.command = 2;
00016     frame.value = "test_value";
00017     frame.unit = "V";
00018     frame.footer = FRAME_END;
00019     return frame;
00020 }
00021
00022 #endif
```

8.125 test/comms/test_frame_send.cpp File Reference

```
#include "unity.h"
#include "communication.h"
#include "../mocks/hardware_mocks.h"
```

Functions

- void setUp (void)
- void tearDown (void)
- void test_send_frame_uart (void)

8.125.1 Function Documentation

8.125.1.1 setUp()

```
void setUp (
            void )
```

Definition at line 6 of file test_frame_send.cpp.

8.125.1.2 tearDown()

```
void tearDown (
            void )
```

Definition at line 12 of file test_frame_send.cpp.

8.125.1.3 test_send_frame_uart()

```
void test_send_frame_uart (
            void )
```

Definition at line 17 of file test_frame_send.cpp.

8.126 test_frame_send.cpp

Go to the documentation of this file.

```
// test/comms/test_frame_send.cpp
#include "unity.h"
#include "communication.h"
#include "../mocks/hardware_mocks.h"

void setUp(void) {
    // Enable mocks before each test
    mock_uart_enabled = true;
    uart_output_buffer.clear();
}

void tearDown(void) {
    // Disable mocks after each test
    mock_uart_enabled = false;
}

void test_send_frame_uart(void) {
    // Create a test frame
    Frame test_frame = {
        .operationType = OperationType::VAL,
        .group = 1,
        .command = 2,
        .value = "TEST_VALUE"
    };

    // Call function under test
    send_frame_uart(test_frame);

    // Verify output using mocks
    TEST_ASSERT_EQUAL(1, uart_output_buffer.size());
    TEST_ASSERT_TRUE(uart_output_buffer[0].find("KBST;0;VAL;1;2;TEST_VALUE;") != std::string::npos);
}
```

8.127 test/mocks/hardware_mocks.cpp File Reference

```
#include "hardware_mocks.h"
#include <cstring>
```

Functions

- void mock_uart_puts (uart_inst_t *uart, const char *str)
- void mock_uart_init (uart_inst_t *uart, uint baudrate)
- void mock_spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)
- int mock_spi_read_blocking (spi_inst_t *spi, uint8_t tx_data, uint8_t *dst, size_t len)

Variables

- bool mock_uart_enabled = false
- std::vector< std::string > uart_output_buffer
- bool mock_spi_enabled = false
- std::vector< uint8_t > spi_output_buffer

8.127.1 Function Documentation

8.127.1.1 mock_uart_puts()

```
void mock_uart_puts (
        uart_inst_t * uart,
        const char * str)
```

Definition at line 9 of file hardware_mocks.cpp.

8.127.1.2 mock_uart_init()

```
void mock_uart_init (
            uart_inst_t * uart,
            uint baudrate)
```

Definition at line 17 of file hardware_mocks.cpp.

8.127.1.3 mock_spi_write_blocking()

```
void mock_spi_write_blocking (
            spi_inst_t * spi,
            const uint8_t * src,
            size_t len)
```

Definition at line 27 of file hardware_mocks.cpp.

8.127.1.4 mock_spi_read_blocking()

```
int mock_spi_read_blocking (
            spi_inst_t * spi,
            uint8_t tx_data,
            uint8_t * dst,
            size_t len)
```

Definition at line 35 of file hardware_mocks.cpp.

8.127.2 Variable Documentation

8.127.2.1 mock_uart_enabled

```
bool mock_uart_enabled = false
```

Definition at line 6 of file hardware_mocks.cpp.

8.127.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer
```

Definition at line 7 of file hardware_mocks.cpp.

8.127.2.3 mock_spi_enabled

```
bool mock_spi_enabled = false
```

Definition at line 24 of file hardware_mocks.cpp.

8.127.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer
```

Definition at line 25 of file hardware_mocks.cpp.

8.128 hardware_mocks.cpp

Go to the documentation of this file.

```
00001 // test/mocks/hardware_mocks.cpp
00002 #include "hardware_mocks.h"
00003 #include <cstring>
00004
00005 // UART mocks
00006 bool mock_uart_enabled = false;
00007 std::vector<std::string> uart_output_buffer;
00008
00009 void mock_uart_puts(uart_inst_t* uart, const char* str) {
00010     if (mock_uart_enabled) {
00011         uart_output_buffer.push_back(std::string(str));
00012     } else {
00013         uart_puts(uart, str);
00014     }
00015 }
00016
00017 void mock_uart_init(uart_inst_t* uart, uint baudrate) {
00018     if (!mock_uart_enabled) {
00019         uart_init(uart, baudrate);
00020     }
00021 }
00022
00023 // SPI mocks
00024 bool mock_spi_enabled = false;
00025 std::vector<uint8_t> spi_output_buffer;
00026
00027 void mock_spi_write_blocking(spi_inst_t* spi, const uint8_t* src, size_t len) {
00028     if (mock_spi_enabled) {
00029         spi_output_buffer.insert(spi_output_buffer.end(), src, src + len);
00030     } else {
00031         spi_write_blocking(spi, src, len);
00032     }
00033 }
00034
00035 int mock_spi_read_blocking(spi_inst_t* spi, uint8_t tx_data, uint8_t* dst, size_t len) {
00036     if (mock_spi_enabled) {
00037         // Mock implementation that fills dst with test data
00038         memset(dst, tx_data, len);
00039         return len;
00040     } else {
00041         return spi_read_blocking(spi, tx_data, dst, len);
00042     }
00043 }
```

8.129 test/mocks/hardware_mocks.h File Reference

```
#include <vector>
#include <string>
```

Functions

- void mock_uart_puts (uart_inst_t *uart, const char *str)
- void mock_uart_init (uart_inst_t *uart, uint baudrate)
- void mock_spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)
- int mock_spi_read_blocking (spi_inst_t *spi, uint8_t tx_data, uint8_t *dst, size_t len)

Variables

- bool mock_uart_enabled
- std::vector< std::string > uart_output_buffer
- bool mock_spi_enabled
- std::vector< uint8_t > spi_output_buffer

8.129.1 Function Documentation

8.129.1.1 mock_uart_puts()

```
void mock_uart_puts (
            uart_inst_t * uart,
            const char * str)
```

Definition at line 9 of file hardware_mocks.cpp.

8.129.1.2 mock_uart_init()

```
void mock_uart_init (
            uart_inst_t * uart,
            uint baudrate)
```

Definition at line 17 of file hardware_mocks.cpp.

8.129.1.3 mock_spi_write_blocking()

```
void mock_spi_write_blocking (
            spi_inst_t * spi,
            const uint8_t * src,
            size_t len)
```

Definition at line 27 of file hardware_mocks.cpp.

8.129.1.4 mock_spi_read_blocking()

```
int mock_spi_read_blocking (
            spi_inst_t * spi,
            uint8_t tx_data,
            uint8_t * dst,
            size_t len)
```

Definition at line 35 of file hardware_mocks.cpp.

8.129.2 Variable Documentation

8.129.2.1 mock_uart_enabled

```
bool mock_uart_enabled [extern]
```

Definition at line 6 of file hardware_mocks.cpp.

8.129.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer [extern]
```

Definition at line 7 of file hardware_mocks.cpp.

8.129.2.3 mock_spi_enabled

```
bool mock_spi_enabled [extern]
```

Definition at line 24 of file hardware_mocks.cpp.

8.129.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer [extern]
```

Definition at line 25 of file hardware_mocks.cpp.

8.130 hardware_mocks.h

Go to the documentation of this file.

```
// test/mocks/hardware_mocks.h
#ifndef HARDWARE_MOCKS_H
#define HARDWARE_MOCKS_H

#include <vector>
#include <string>

// UART mocks
extern bool mock_uart_enabled;
extern std::vector<std::string> uart_output_buffer;

void mock_uart_puts(uart_inst_t* uart, const char* str);
void mock_uart_init(uart_inst_t* uart, uint baudrate);

// SPI mocks
extern bool mock_spi_enabled;
extern std::vector<uint8_t> spi_output_buffer;

void mock_spi_write_blocking(spi_inst_t* spi, const uint8_t* src, size_t len);
int mock_spi_read_blocking(spi_inst_t* spi, uint8_t tx_data, uint8_t* dst, size_t len);

#endif // HARDWARE_MOCKS_H
```

8.131 test/test_runner.cpp File Reference

```
#include "includes.h"
#include "unity.h"
```

Functions

- void test_frame_encode_basic (void)
- void test_frame_decode_basic (void)
- void test_frame_decode_invalid_header (void)
- void test_frame_build_get (void)
- void test_frame_build_set (void)
- void test_frame_build_res (void)
- void test_frame_build_seq (void)
- void test_frame_build_val (void)
- void test_frame_build_err (void)
- void test_operation_type_conversion (void)
- void test_value_unit_type_conversion (void)
- void test_exception_type_conversion (void)
- void test_hex_string_conversion (void)
- void test_command_handler_get_operation (void)
- void test_command_handler_set_operation (void)
- void test_command_handler_invalid_operation (void)
- void test_handle_get_commands_list (void)
- void test_handle_get_build_version (void)
- void test_handle_verbosity (void)
- void test_handle_enter_bootloader_mode (void)
- void test_error_code_conversion (void)
- int main (void)

8.131.1 Function Documentation

8.131.1.1 test_frame_encode_basic()

```
void test_frame_encode_basic (
             void )  [extern]
```

Definition at line 4 of file test_frame_coding.cpp.

8.131.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
             void )  [extern]
```

Definition at line 16 of file test_frame_coding.cpp.

8.131.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
             void )  [extern]
```

Definition at line 29 of file test_frame_coding.cpp.

8.131.1.4 test_frame_build_get()

```
void test_frame_build_get (
            void ) [extern]
```

Definition at line 24 of file test_frame_build.cpp.

8.131.1.5 test_frame_build_set()

```
void test_frame_build_set (
            void ) [extern]
```

Definition at line 35 of file test_frame_build.cpp.

8.131.1.6 test_frame_build_res()

```
void test_frame_build_res (
            void ) [extern]
```

Definition at line 46 of file test_frame_build.cpp.

8.131.1.7 test_frame_build_seq()

```
void test_frame_build_seq (
            void ) [extern]
```

Definition at line 57 of file test_frame_build.cpp.

8.131.1.8 test_frame_build_val()

```
void test_frame_build_val (
            void ) [extern]
```

Definition at line 4 of file test_frame_build.cpp.

8.131.1.9 test_frame_build_err()

```
void test_frame_build_err (
            void ) [extern]
```

Definition at line 15 of file test_frame_build.cpp.

8.131.1.10 test_operation_type_conversion()

```
void test_operation_type_conversion (
            void ) [extern]
```

Definition at line 4 of file test_converters.cpp.

8.131.1.11 test_value_unit_type_conversion()

```
void test_value_unit_type_conversion (
            void )  [extern]
```

Definition at line 13 of file test_converters.cpp.

8.131.1.12 test_exception_type_conversion()

```
void test_exception_type_conversion (
            void )  [extern]
```

Definition at line 20 of file test_converters.cpp.

8.131.1.13 test_hex_string_conversion()

```
void test_hex_string_conversion (
            void )  [extern]
```

Definition at line 27 of file test_converters.cpp.

8.131.1.14 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
            void )  [extern]
```

Definition at line 29 of file test_comand_handlers.cpp.

8.131.1.15 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
            void )  [extern]
```

Definition at line 39 of file test_comand_handlers.cpp.

8.131.1.16 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
            void )  [extern]
```

Definition at line 54 of file test_comand_handlers.cpp.

8.131.1.17 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
            void )  [extern]
```

Definition at line 6 of file test_diagnostic_commands.cpp.

8.131.1.18 test_handle_get_build_version()

```
void test_handle_get_build_version (
            void ) [extern]
```

Definition at line 15 of file test_diagnostic_commands.cpp.

8.131.1.19 test_handle_verbosity()

```
void test_handle_verbosity (
            void ) [extern]
```

Definition at line 25 of file test_diagnostic_commands.cpp.

8.131.1.20 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
            void ) [extern]
```

Definition at line 40 of file test_diagnostic_commands.cpp.

8.131.1.21 test_error_code_conversion()

```
void test_error_code_conversion (
            void ) [extern]
```

8.131.1.22 main()

```
int main (
            void )
```

Definition at line 38 of file test_runner.cpp.

8.132 test_runner.cpp

Go to the documentation of this file.

```
// test/test_runner.cpp
#include "includes.h"
#include "unity.h"

// External test function declarations
// Pure software tests (no hardware dependencies)
extern void test_frame_encode_basic(void);
extern void test_frame_decode_basic(void);
extern void test_frame_decode_invalid_header(void);

// FRAME BUILD
extern void test_frame_build_get(void);
extern void test_frame_build_set(void);
extern void test_frame_build_res(void);
extern void test_frame_build_seq(void);
extern void test_frame_build_val(void);
extern void test_frame_build_err(void);

```

```
extern void test_operation_type_conversion(void);
extern void test_value_unit_type_conversion(void);
extern void test_exception_type_conversion(void);
extern void test_hex_string_conversion(void);

// Command handler tests
extern void test_command_handler_get_operation(void);
extern void test_command_handler_set_operation(void);
extern void test_command_handler_invalid_operation(void);

//diagnostic
extern void test_handle_get_commands_list(void);
extern void test_handle_get_build_version(void);
extern void test_handle_verbosity(void);
extern void test_handle_enter_bootloader_mode(void);

// Error code tests
extern void test_error_code_conversion(void);

int main(void) {
    stdio_init_all();
    uart_init(uart0, 115200);
    gpio_set_function(0, GPIO_FUNC_UART);
    gpio_set_function(1, GPIO_FUNC_UART);

    UNITY_BEGIN();
    uart_puts(uart0, "begin unity tests\n");

    // Frame codec tests (pure software)
    uart_puts(uart0, "begin frame codec tests\n");
    RUN_TEST(test_frame_encode_basic);
    RUN_TEST(test_frame_decode_basic);
    RUN_TEST(test_frame_decode_invalid_header);
    uart_puts(uart0, "end frame codec tests\n");

    // Frame build tests (pure software)
    uart_puts(uart0, "begin frame build tests\n");
    RUN_TEST(test_frame_build_get);
    RUN_TEST(test_frame_build_set);
    RUN_TEST(test_frame_build_res);
    RUN_TEST(test_frame_build_seq);
    uart_puts(uart0, "end frame build tests\n");

    // Converter tests (pure software)
    uart_puts(uart0, "begin converter tests\n");
    RUN_TEST(test_operation_type_conversion);
    RUN_TEST(test_value_unit_type_conversion);
    RUN_TEST(test_exception_type_conversion);
    RUN_TEST(test_hex_string_conversion);
    RUN_TEST(test_error_code_conversion);
    uart_puts(uart0, "end converter tests\n");

    // Command handler tests (pure software)
    uart_puts(uart0, "begin command handler tests\n");
    RUN_TEST(test_command_handler_get_operation);
    RUN_TEST(test_command_handler_set_operation);
    RUN_TEST(test_command_handler_invalid_operation);
    uart_puts(uart0, "end command handler tests\n");

    uart_puts(uart0, "begin diagnostic command handlers tests\n");
    RUN_TEST(test_handle_get_commands_list);
    RUN_TEST(test_handle_get_build_version);
    RUN_TEST(test_handle_verbosity);
    RUN_TEST(test_handle_enter_bootloader_mode);
    uart_puts(uart0, "end diagnostic commands handlers tests\n");

    return UNITY_END();
}
```

Index