KubiSat Firmware

Generated by Doxygen 1.13.2

1 refactor 1

1.0.1 Singleton Implementation Analysis . . . 1
1.0.2 Recommended Improvements . . . 2
1.0.3 Static Analysis Tools . . . 2
1.0.4 Configuration Example . . . 3
1.0.5 Analysis of Current Singleton Implementations . . . 3
1.0.6 Recommendations for Singleton Implementation . . . 3
1.0.6.1 1. Standardize on Local Static Instance Pattern . . . 3
1.0.6.2 2. Add Proper Construction Requirements . . . 4
1.0.6.3 3. Use Static Analysis Tools . . . 4
1.0.7 Implementation Example . . . 4
1.0.8 Alternatives to Consider . . . 4
1.0.9 Design Patterns to Consider . . . 5
1.0.9.1 1. Dependency Injection Pattern . . . 5
1.0.9.2 2. Factory Method Pattern . . . 5
1.0.9.3 3. Observer Pattern . . . 6
1.0.9.4 4. Command Pattern . . . 6
1.0.9.5 5. Builder Pattern . . . 6
1.0.10 Architectural Improvements . . . 7
1.0.10.1 1. Service Locator Pattern . . . 7
1.0.10.2 2. Repository Pattern . . . 7
1.0.10.3 3. RAII for Resource Management . . . 7
1.0.11 Practical Implementation Plan . . . 8

2 todo 9

3 Clock Commands 13

4 Topic Index 15

4.1 Topics . . . 15

5 Hierarchical Index 17

5.1 Class Hierarchy . . . 17

6 Class Index 19

6.1 Class List . . . 19

7 File Index 21

7.1 File List . . . 21

8 Topic Documentation 23

8.1 Clock Management Commands . . . 23
8.1.1 Detailed Description . . . 23
8.1.2 Function Documentation . . . 23
8.1.2.1 handle_time() . . . 23

8.1.2.2 handle_timezone_offset() . . . 24
8.1.2.3 handle_clock_sync_interval() . . . 25
8.1.2.4 handle_get_last_sync_time() . . . 26
8.2 Command System . . . 27
8.2.1 Detailed Description . . . 27
8.2.2 Typedef Documentation . . . 27
8.2.2.1 CommandHandler . . . 27
8.2.2.2 CommandMap . . . 27
8.2.3 Function Documentation . . . 27
8.2.3.1 execute_command() . . . 27
8.2.4 Variable Documentation . . . 28
8.2.4.1 command_handlers . . . 28
8.3 Diagnostic Commands . . . 29
8.3.1 Detailed Description . . . 29
8.3.2 Function Documentation . . . 29
8.3.2.1 handle_get_commands_list() . . . 29
8.3.2.2 handle_get_build_version() . . . 30
8.3.2.3 handle_verbosity() . . . 31
8.3.2.4 handle_enter_bootloader_mode() . . . 33
8.4 Event Commands . . . 34
8.4.1 Detailed Description . . . 35
8.4.2 Function Documentation . . . 35
8.4.2.1 handle_get_last_events() . . . 35
8.4.2.2 handle_get_event_count() . . . 36
8.5 GPS Commands . . . 37
8.5.1 Detailed Description . . . 37
8.5.2 Function Documentation . . . 37
8.5.2.1 handle_gps_power_status() . . . 37
8.5.2.2 handle_enable_gps_uart_passthrough() . . . 38
8.5.2.3 handle_get_rmc_data() . . . 39
8.5.2.4 handle_get_gga_data() . . . 40
8.6 Power Commands . . . 41
8.6.1 Detailed Description . . . 41
8.6.2 Function Documentation . . . 41
8.6.2.1 handle_get_power_manager_ids() . . . 41
8.6.2.2 handle_get_voltage_battery() . . . 42
8.6.2.3 handle_get_voltage_5v() . . . 43
8.6.2.4 handle_get_current_charge_usb() . . . 44
8.6.2.5 handle_get_current_charge_solar() . . . 45
8.6.2.6 handle_get_current_charge_total() . . . 46
8.6.2.7 handle_get_current_draw() . . . 47
8.7 Sensor Commands . . . 48

8.7.1 Detailed Description . . . 48
8.7.2 Function Documentation . . . 48
8.7.2.1 handle_get_sensor_data() . . . 48
8.7.2.2 handle_sensor_config() . . . 49
8.7.2.3 handle_get_sensor_list() . . . 50
8.8 Storage Commands . . . 51
8.8.1 Detailed Description . . . 52
8.8.2 Function Documentation . . . 52
8.8.2.1 handle_list_files() . . . 52
8.8.2.2 handle_mount() . . . 52
8.9 Telemetry Buffer Commands . . . 53
8.9.1 Detailed Description . . . 54
8.9.2 Function Documentation . . . 54
8.9.2.1 handle_get_last_telemetry_record() . . . 54
8.9.2.2 handle_get_last_sensor_record() . . . 55
8.10 Frame Handling . . . 55
8.10.1 Detailed Description . . . 56
8.10.2 Function Documentation . . . 56
8.10.2.1 frame_encode() . . . 56
8.10.2.2 frame_decode() . . . 57
8.10.2.3 frame_process() . . . 58
8.10.2.4 frame_build() . . . 59
8.11 Protocol . . . 60
8.11.1 Detailed Description . . . 61
8.11.2 Enumeration Type Documentation . . . 61
8.11.2.1 ErrorCode . . . 61
8.11.2.2 OperationType . . . 62
8.11.2.3 CommandAccessLevel . . . 62
8.11.2.4 ValueUnit . . . 62
8.11.2.5 ExceptionType . . . 63
8.11.2.6 Interface . . . 63
8.12 Receiving Data . . . 63
8.12.1 Detailed Description . . . 64
8.12.2 Function Documentation . . . 64
8.12.2.1 on_receive() . . . 64
8.12.2.2 handle_uart_input() . . . 65
8.13 Utility Converters . . . 65
8.13.1 Detailed Description . . . 66
8.13.2 Function Documentation . . . 66
8.13.2.1 exception_type_to_string() . . . 66
8.13.2.2 value_unit_type_to_string() . . . 66
8.13.2.3 operation_type_to_string() . . . 68

8.13.2.4 string_to_operation_type() . . . 68
8.13.2.5 error_code_to_string() . . . 69
8.13.2.6 hex_string_to_bytes() . . . 70
8.14 RTC clock . . . 71
8.14.1 Detailed Description . . . 72
8.14.2 Function Documentation . . . 72
8.14.2.1 DS3231() . . . 72
8.14.2.2 get_instance() . . . 73
8.14.2.3 set_time() . . . 73
8.14.2.4 get_time() . . . 74
8.14.2.5 read_temperature() . . . 76
8.14.2.6 set_unix_time() . . . 77
8.14.2.7 get_unix_time() . . . 78
8.14.2.8 clock_enable() . . . 79
8.14.2.9 get_timezone_offset() . . . 80
8.14.2.10 set_timezone_offset() . . . 81
8.14.2.11 get_clock_sync_interval() . . . 82
8.14.2.12 set_clock_sync_interval() . . . 82
8.14.2.13 get_last_sync_time() . . . 83
8.14.2.14 update_last_sync_time() . . . 84
8.14.2.15 get_local_time() . . . 84
8.14.2.16 is_sync_needed() . . . 85
8.14.2.17 sync_clock_with_gps() . . . 86
8.14.2.18 i2c_read_reg() . . . 87
8.14.2.19 i2c_write_reg() . . . 88
8.14.2.20 bin_to_bcd() . . . 90
8.14.2.21 bcd_to_bin() . . . 91
8.15 Event Management . . . 92
8.15.1 Detailed Description . . . 93
8.15.2 Macro Definition Documentation . . . 94
8.15.2.1 EVENT_BUFFER_SIZE . . . 94
8.15.2.2 EVENT_FLUSH_THRESHOLD . . . 94
8.15.2.3 EVENT_LOG_FILE . . . 94
8.15.3 Enumeration Type Documentation . . . 94
8.15.3.1 EventGroup . . . 94
8.15.3.2 SystemEvent . . . 94
8.15.3.3 PowerEvent . . . 95
8.15.3.4 CommsEvent . . . 95
8.15.3.5 GPSEvent . . . 95
8.15.3.6 ClockEvent . . . 96
8.15.4 Function Documentation . . . 96
8.15.4.1 __attribute__() . . . 96

8.15.4.2 init() . . . 96
8.15.4.3 log_event() . . . 97
8.15.4.4 get_event() . . . 98
8.15.4.5 save_to_storage() . . . 98
8.15.5 Variable Documentation . . . 99
8.15.5.1 __attribute__ . . . 99
8.16 Location . . . 99
8.16.1 Detailed Description . . . 100
8.16.2 Macro Definition Documentation . . . 100
8.16.2.1 MAX_RAW_DATA_LENGTH . . . 100
8.16.3 Function Documentation . . . 100
8.16.3.1 splitString() . . . 100
8.16.3.2 collect_gps_data() . . . 101
8.17 INA3221 Power Monitor . . . 102
8.17.1 Detailed Description . . . 102
8.17.2 Configuration Functions . . . 102
8.17.2.1 Detailed Description . . . 103
8.17.2.2 Function Documentation . . . 103
8.17.3 Measurement Functions . . . 112
8.17.3.1 Detailed Description . . . 112
8.17.3.2 Function Documentation . . . 112
8.18 Power Management . . . 114
8.18.1 Detailed Description . . . 115
8.18.2 Function Documentation . . . 115
8.18.2.1 PowerManager() . . . 115
8.18.2.2 get_instance() . . . 115
8.18.2.3 initialize() . . . 116
8.18.2.4 read_device_ids() . . . 117
8.18.2.5 get_voltage_battery() . . . 117
8.18.2.6 get_voltage_5v() . . . 118
8.18.2.7 get_current_charge_usb() . . . 118
8.18.2.8 get_current_draw() . . . 119
8.18.2.9 get_current_charge_solar() . . . 119
8.18.2.10 get_current_charge_total() . . . 120
8.18.2.11 configure() . . . 120
8.18.2.12 is_charging_solar() . . . 121
8.18.2.13 is_charging_usb() . . . 121
8.19 Sensors . . . 121
8.19.1 Detailed Description . . . 122
8.19.2 Enumeration Type Documentation . . . 122
8.19.2.1 SensorType . . . 122
8.19.2.2 SensorDataTypeIdentifier . . . 123

8.19.3 Function Documentation . . . 123
8.19.3.1 sensor_init() . . . 123
8.19.3.2 sensor_configure() . . . 123
8.19.3.3 sensor_read_data() . . . 124
8.19.3.4 get_sensor() . . . 125
8.19.3.5 scan_connected_sensors() . . . 125
8.19.3.6 get_available_sensors() . . . 126
8.20 Storage . . . 126
8.20.1 Detailed Description . . . 126
8.20.2 Function Documentation . . . 127
8.20.2.1 fs_init() . . . 127
8.20.2.2 fs_stop() . . . 128
8.21 System State Manager . . . 128
8.21.1 Detailed Description . . . 128
8.22 Telemetry Manager . . . 128
8.22.1 Detailed Description . . . 130
8.22.2 Macro Definition Documentation . . . 130
8.22.2.1 TELEMETRY_CSV_PATH . . . 130
8.22.2.2 SENSOR_DATA_CSV_PATH . . . 130
8.22.2.3 DEFAULT_SAMPLE_INTERVAL_MS . . . 131
8.22.2.4 DEFAULT_FLUSH_THRESHOLD . . . 131
8.22.3 Function Documentation . . . 131
8.22.3.1 telemetry_init() . . . 131
8.22.3.2 collect_power_telemetry() . . . 132
8.22.3.3 emit_power_events() . . . 132
8.22.3.4 collect_gps_telemetry() . . . 133
8.22.3.5 collect_sensor_telemetry() . . . 134
8.22.3.6 collect_telemetry() . . . 135
8.22.3.7 flush_telemetry() . . . 135
8.22.3.8 flush_sensor_data() . . . 136
8.22.3.9 is_telemetry_collection_time() . . . 137
8.22.3.10 is_telemetry_flush_time() . . . 138
8.22.3.11 get_last_telemetry_record_csv() . . . 138
8.22.3.12 get_last_sensor_record_csv() . . . 139
8.22.3.13 to_csv() [1/2] . . . 140
8.22.3.14 to_csv() [2/2] . . . 140
8.22.4 Variable Documentation . . . 140
8.22.4.1 sample_interval_ms . . . 140
8.22.4.2 flush_threshold . . . 141
8.22.4.3 telemetry_buffer [1/2] . . . 141
8.22.4.4 telemetry_buffer_count [1/2] . . . 141
8.22.4.5 telemetry_buffer_write_index [1/2] . . . 141

8.22.4.6 sensor_data_buffer [1/2] . . . 141
8.22.4.7 sensor_data_buffer_count [1/2] . . . 141
8.22.4.8 sensor_data_buffer_write_index [1/2] . . . 141
8.22.4.9 telemetry_mutex [1/2] . . . 142
8.22.4.10 TELEMETRY_BUFFER_SIZE [1/2] . . . 142
8.22.4.11 telemetry_buffer [2/2] . . . 142
8.22.4.12 telemetry_buffer_count [2/2] . . . 142
8.22.4.13 telemetry_buffer_write_index [2/2] . . . 142
8.22.4.14 sensor_data_buffer [2/2] . . . 142
8.22.4.15 sensor_data_buffer_count [2/2] . . . 142
8.22.4.16 sensor_data_buffer_write_index [2/2] . . . 143
8.22.4.17 telemetry_mutex [2/2] . . . 143
8.22.4.18 TELEMETRY_BUFFER_SIZE [2/2] . . . 143

9 Class Documentation **145**
9.1 BH1750 Class Reference . . . 145
9.1.1 Detailed Description . . . 145
9.1.2 Member Enumeration Documentation . . . 145
9.1.2.1 Mode . . . 145
9.1.3 Constructor & Destructor Documentation . . . 146
9.1.3.1 BH1750() . . . 146
9.1.4 Member Function Documentation . . . 146
9.1.4.1 begin() . . . 146
9.1.4.2 configure() . . . 147
9.1.4.3 get_light_level() . . . 147
9.1.4.4 write8() . . . 147
9.1.5 Member Data Documentation . . . 148
9.1.5.1 _i2c_addr . . . 148
9.1.5.2 i2c_port_ . . . 148
9.2 BH1750Wrapper Class Reference . . . 148
9.2.1 Detailed Description . . . 149
9.2.2 Constructor & Destructor Documentation . . . 149
9.2.2.1 BH1750Wrapper() [1/2] . . . 149
9.2.2.2 BH1750Wrapper() [2/2] . . . 149
9.2.3 Member Function Documentation . . . 150
9.2.3.1 get_i2c_addr() . . . 150
9.2.3.2 init() . . . 150
9.2.3.3 read_data() . . . 150
9.2.3.4 is_initialized() . . . 151
9.2.3.5 get_type() . . . 151
9.2.3.6 configure() . . . 151
9.2.3.7 get_address() . . . 152

9.2.4 Member Data Documentation . . . 152
9.2.4.1 sensor_ . . . 152
9.2.4.2 initialized_ . . . 152
9.2.4.3 i2c_port_ . . . 152
9.3 BME280 Class Reference . . . 153
9.3.1 Detailed Description . . . 154
9.3.2 Constructor & Destructor Documentation . . . 154
9.3.2.1 BME280() . . . 154
9.3.3 Member Function Documentation . . . 155
9.3.3.1 init() . . . 155
9.3.3.2 reset() . . . 155
9.3.3.3 read_raw_all() . . . 155
9.3.3.4 convert_temperature() . . . 155
9.3.3.5 convert_pressure() . . . 155
9.3.3.6 convert_humidity() . . . 156
9.3.3.7 configure_sensor() . . . 156
9.3.3.8 get_calibration_parameters() . . . 156
9.3.4 Member Data Documentation . . . 156
9.3.4.1 ADDR_SDO_LOW . . . 156
9.3.4.2 ADDR_SDO_HIGH . . . 157
9.3.4.3 i2c_port . . . 157
9.3.4.4 device_addr . . . 157
9.3.4.5 calib_params . . . 157
9.3.4.6 initialized_ . . . 157
9.3.4.7 t_fine . . . 157
9.3.4.8 REG_CONFIG . . . 157
9.3.4.9 REG_CTRL_MEAS . . . 157
9.3.4.10 REG_CTRL_HUM . . . 158
9.3.4.11 REG_RESET . . . 158
9.3.4.12 REG_PRESSURE_MSB . . . 158
9.3.4.13 REG_TEMPERATURE_MSB . . . 158
9.3.4.14 REG_HUMIDITY_MSB . . . 158
9.3.4.15 REG_DIG_T1_LSB . . . 158
9.3.4.16 REG_DIG_T1_MSB . . . 158
9.3.4.17 REG_DIG_T2_LSB . . . 158
9.3.4.18 REG_DIG_T2_MSB . . . 159
9.3.4.19 REG_DIG_T3_LSB . . . 159
9.3.4.20 REG_DIG_T3_MSB . . . 159
9.3.4.21 REG_DIG_P1_LSB . . . 159
9.3.4.22 REG_DIG_P1_MSB . . . 159
9.3.4.23 REG_DIG_P2_LSB . . . 159
9.3.4.24 REG_DIG_P2_MSB . . . 159

9.3.4.25 REG_DIG_P3_LSB . . . 159
9.3.4.26 REG_DIG_P3_MSB . . . 160
9.3.4.27 REG_DIG_P4_LSB . . . 160
9.3.4.28 REG_DIG_P4_MSB . . . 160
9.3.4.29 REG_DIG_P5_LSB . . . 160
9.3.4.30 REG_DIG_P5_MSB . . . 160
9.3.4.31 REG_DIG_P6_LSB . . . 160
9.3.4.32 REG_DIG_P6_MSB . . . 160
9.3.4.33 REG_DIG_P7_LSB . . . 160
9.3.4.34 REG_DIG_P7_MSB . . . 161
9.3.4.35 REG_DIG_P8_LSB . . . 161
9.3.4.36 REG_DIG_P8_MSB . . . 161
9.3.4.37 REG_DIG_P9_LSB . . . 161
9.3.4.38 REG_DIG_P9_MSB . . . 161
9.3.4.39 REG_DIG_H1 . . . 161
9.3.4.40 REG_DIG_H2 . . . 161
9.3.4.41 REG_DIG_H3 . . . 161
9.3.4.42 REG_DIG_H4 . . . 162
9.3.4.43 REG_DIG_H5 . . . 162
9.3.4.44 REG_DIG_H6 . . . 162
9.3.4.45 NUM_CALIB_PARAMS . . . 162
9.4 BME280CalibParam Struct Reference . . . 162
9.4.1 Detailed Description . . . 163
9.4.2 Member Data Documentation . . . 163
9.4.2.1 dig_t1 . . . 163
9.4.2.2 dig_t2 . . . 163
9.4.2.3 dig_t3 . . . 163
9.4.2.4 dig_p1 . . . 163
9.4.2.5 dig_p2 . . . 163
9.4.2.6 dig_p3 . . . 163
9.4.2.7 dig_p4 . . . 164
9.4.2.8 dig_p5 . . . 164
9.4.2.9 dig_p6 . . . 164
9.4.2.10 dig_p7 . . . 164
9.4.2.11 dig_p8 . . . 164
9.4.2.12 dig_p9 . . . 164
9.4.2.13 dig_h1 . . . 164
9.4.2.14 dig_h2 . . . 164
9.4.2.15 dig_h3 . . . 165
9.4.2.16 dig_h4 . . . 165
9.4.2.17 dig_h5 . . . 165
9.4.2.18 dig_h6 . . . 165

9.5 BME280Wrapper Class Reference . . . 165
9.5.1 Detailed Description . . . 166
9.5.2 Constructor & Destructor Documentation . . . 167
9.5.2.1 BME280Wrapper() . . . 167
9.5.3 Member Function Documentation . . . 167
9.5.3.1 init() . . . 167
9.5.3.2 read_data() . . . 167
9.5.3.3 is_initialized() . . . 168
9.5.3.4 get_type() . . . 168
9.5.3.5 configure() . . . 168
9.5.3.6 get_address() . . . 169
9.5.4 Member Data Documentation . . . 169
9.5.4.1 sensor_ . . . 169
9.5.4.2 initialized_ . . . 169
9.6 INA3221::conf_reg_t Struct Reference . . . 169
9.6.1 Detailed Description . . . 170
9.6.2 Member Data Documentation . . . 170
9.6.2.1 mode_shunt_en . . . 170
9.6.2.2 mode_bus_en . . . 170
9.6.2.3 mode_continious_en . . . 170
9.6.2.4 shunt_conv_time . . . 170
9.6.2.5 bus_conv_time . . . 171
9.6.2.6 avg_mode . . . 171
9.6.2.7 ch3_en . . . 171
9.6.2.8 ch2_en . . . 171
9.6.2.9 ch1_en . . . 171
9.6.2.10 reset . . . 171
9.7 DS3231 Class Reference . . . 171
9.7.1 Detailed Description . . . 173
9.7.2 Constructor & Destructor Documentation . . . 173
9.7.2.1 DS3231() [1/2] . . . 173
9.7.2.2 DS3231() [2/2] . . . 174
9.7.3 Member Function Documentation . . . 174
9.7.3.1 operator=() . . . 174
9.7.4 Member Data Documentation . . . 175
9.7.4.1 i2c . . . 175
9.7.4.2 ds3231_addr . . . 175
9.7.4.3 clock_mutex_ . . . 175
9.7.4.4 timezone_offset_minutes_ . . . 175
9.7.4.5 sync_interval_minutes_ . . . 175
9.7.4.6 last_sync_time_ . . . 175
9.8 ds3231_data_t Struct Reference . . . 176

9.8.1 Detailed Description . . . 176
9.8.2 Member Data Documentation . . . 176
9.8.2.1 seconds . . . 176
9.8.2.2 minutes . . . 176
9.8.2.3 hours . . . 177
9.8.2.4 day . . . 177
9.8.2.5 date . . . 177
9.8.2.6 month . . . 177
9.8.2.7 year . . . 177
9.8.2.8 century . . . 177
9.9 EventEmitter Class Reference . . . 178
9.9.1 Detailed Description . . . 178
9.9.2 Member Function Documentation . . . 178
9.9.2.1 emit() . . . 178
9.10 EventLog Class Reference . . . 179
9.10.1 Detailed Description . . . 179
9.10.2 Member Data Documentation . . . 180
9.10.2.1 id . . . 180
9.10.2.2 timestamp . . . 180
9.10.2.3 group . . . 180
9.10.2.4 event . . . 180
9.11 EventManager Class Reference . . . 180
9.11.1 Detailed Description . . . 181
9.11.2 Constructor & Destructor Documentation . . . 182
9.11.2.1 EventManager() [1/2] . . . 182
9.11.2.2 EventManager() [2/2] . . . 182
9.11.3 Member Function Documentation . . . 182
9.11.3.1 operator=() . . . 182
9.11.3.2 get_instance() . . . 183
9.11.3.3 get_event_count() . . . 183
9.11.3.4 load_from_storage() . . . 184
9.11.4 Member Data Documentation . . . 184
9.11.4.1 events . . . 184
9.11.4.2 eventCount . . . 184
9.11.4.3 writeIndex . . . 184
9.11.4.4 eventMutex . . . 184
9.11.4.5 nextEventId . . . 185
9.11.4.6 eventsSinceFlush . . . 185
9.12 Frame Struct Reference . . . 185
9.12.1 Detailed Description . . . 185
9.12.2 Member Data Documentation . . . 186
9.12.2.1 header . . . 186

9.12.2.2 direction . . . 186
9.12.2.3 operationType . . . 186
9.12.2.4 group . . . 186
9.12.2.5 command . . . 186
9.12.2.6 value . . . 186
9.12.2.7 unit . . . 186
9.12.2.8 footer . . . 187
9.13 INA3221 Class Reference . . . 187
9.13.1 Detailed Description . . . 189
9.13.2 Member Function Documentation . . . 189
9.13.2.1 _read() . . . 189
9.13.2.2 _write() . . . 190
9.13.2.3 get_current() . . . 191
9.13.3 Member Data Documentation . . . 192
9.13.3.1 _i2c_addr . . . 192
9.13.3.2 _i2c . . . 192
9.13.3.3 _shuntRes . . . 192
9.13.3.4 _filterRes . . . 192
9.13.3.5 _masken_reg . . . 192
9.14 ISensor Class Reference . . . 193
9.14.1 Detailed Description . . . 193
9.14.2 Constructor & Destructor Documentation . . . 194
9.14.2.1 ~ISensor() . . . 194
9.14.3 Member Function Documentation . . . 194
9.14.3.1 init() . . . 194
9.14.3.2 read_data() . . . 194
9.14.3.3 is_initialized() . . . 194
9.14.3.4 get_type() . . . 195
9.14.3.5 configure() . . . 195
9.14.3.6 get_address() . . . 195
9.15 INA3221::masken_reg_t Struct Reference . . . 195
9.15.1 Detailed Description . . . 196
9.15.2 Member Data Documentation . . . 196
9.15.2.1 conv_ready . . . 196
9.15.2.2 timing_ctrl_alert . . . 196
9.15.2.3 pwr_valid_alert . . . 196
9.15.2.4 warn_alert_ch3 . . . 196
9.15.2.5 warn_alert_ch2 . . . 197
9.15.2.6 warn_alert_ch1 . . . 197
9.15.2.7 shunt_sum_alert . . . 197
9.15.2.8 crit_alert_ch3 . . . 197
9.15.2.9 crit_alert_ch2 . . . 197

9.15.2.10 crit_alert_ch1 . . . 197
9.15.2.11 crit_alert_latch_en . . . 197
9.15.2.12 warn_alert_latch_en . . . 197
9.15.2.13 shunt_sum_en_ch3 . . . 198
9.15.2.14 shunt_sum_en_ch2 . . . 198
9.15.2.15 shunt_sum_en_ch1 . . . 198
9.15.2.16 reserved . . . 198
9.16 NMEAData Class Reference . . . 198
9.16.1 Detailed Description . . . 199
9.16.2 Constructor & Destructor Documentation . . . 199
9.16.2.1 NMEAData() [1/2] . . . 199
9.16.2.2 NMEAData() [2/2] . . . 200
9.16.3 Member Function Documentation . . . 200
9.16.3.1 operator=() . . . 200
9.16.3.2 get_instance() . . . 200
9.16.3.3 update_rmc_tokens() . . . 201
9.16.3.4 update_gga_tokens() . . . 202
9.16.3.5 get_rmc_tokens() . . . 202
9.16.3.6 get_gga_tokens() . . . 202
9.16.3.7 has_valid_time() . . . 203
9.16.3.8 get_unix_time() . . . 203
9.16.4 Member Data Documentation . . . 203
9.16.4.1 rmc_tokens_ . . . 203
9.16.4.2 gga_tokens_ . . . 204
9.16.4.3 rmc_mutex_ . . . 204
9.16.4.4 gga_mutex_ . . . 204
9.17 PowerManager Class Reference . . . 204
9.17.1 Detailed Description . . . 206
9.17.2 Constructor & Destructor Documentation . . . 206
9.17.2.1 PowerManager() [1/2] . . . 206
9.17.2.2 PowerManager() [2/2] . . . 207
9.17.3 Member Function Documentation . . . 207
9.17.3.1 operator=() . . . 207
9.17.4 Member Data Documentation . . . 208
9.17.4.1 SOLAR_CURRENT_THRESHOLD . . . 208
9.17.4.2 USB_CURRENT_THRESHOLD . . . 208
9.17.4.3 BATTERY_LOW_THRESHOLD . . . 208
9.17.4.4 BATTERY_FULL_THRESHOLD . . . 208
9.17.4.5 ina3221_ . . . 208
9.17.4.6 initialized_ . . . 208
9.17.4.7 powerman_mutex_ . . . 209
9.17.4.8 charging_solar_active_ . . . 209

9.17.4.9 charging_usb_active_ . . . 209
9.18 SensorDataRecord Struct Reference . . . 209
9.18.1 Detailed Description . . . 210
9.18.2 Member Data Documentation . . . 210
9.18.2.1 timestamp . . . 210
9.18.2.2 temperature . . . 210
9.18.2.3 pressure . . . 210
9.18.2.4 humidity . . . 210
9.18.2.5 light . . . 210
9.19 SensorWrapper Class Reference . . . 211
9.19.1 Detailed Description . . . 211
9.19.2 Constructor & Destructor Documentation . . . 212
9.19.2.1 SensorWrapper() . . . 212
9.19.3 Member Function Documentation . . . 212
9.19.3.1 get_instance() . . . 212
9.19.4 Member Data Documentation . . . 213
9.19.4.1 sensors . . . 213
9.20 SystemStateManager Class Reference . . . 213
9.20.1 Detailed Description . . . 214
9.20.2 Constructor & Destructor Documentation . . . 214
9.20.2.1 SystemStateManager() [1/2] . . . 214
9.20.2.2 SystemStateManager() [2/2] . . . 215
9.20.3 Member Function Documentation . . . 216
9.20.3.1 operator=() . . . 216
9.20.3.2 get_instance() . . . 216
9.20.3.3 is_bootloader_reset_pending() . . . 217
9.20.3.4 set_bootloader_reset_pending() . . . 217
9.20.3.5 is_gps_collection_paused() . . . 218
9.20.3.6 set_gps_collection_paused() . . . 218
9.20.3.7 is_sd_card_mounted() . . . 219
9.20.3.8 set_sd_card_mounted() . . . 219
9.20.3.9 get_uart_verbosity() . . . 220
9.20.3.10 set_uart_verbosity() . . . 220
9.20.4 Member Data Documentation . . . 221
9.20.4.1 pending_bootloader_reset . . . 221
9.20.4.2 gps_collection_paused . . . 221
9.20.4.3 sd_card_mounted . . . 221
9.20.4.4 uart_verbosity . . . 221
9.20.4.5 mutex_ . . . 221
9.21 TelemetryRecord Struct Reference . . . 222
9.21.1 Detailed Description . . . 222
9.21.2 Member Data Documentation . . . 222

9.21.2.1 timestamp . . . 222
9.21.2.2 build_version . . . 223
9.21.2.3 battery_voltage . . . 223
9.21.2.4 system_voltage . . . 223
9.21.2.5 charge_current_usb . . . 223
9.21.2.6 charge_current_solar . . . 223
9.21.2.7 discharge_current . . . 223
9.21.2.8 time . . . 224
9.21.2.9 latitude . . . 224
9.21.2.10 lat_dir . . . 224
9.21.2.11 longitude . . . 224
9.21.2.12 lon_dir . . . 224
9.21.2.13 speed . . . 224
9.21.2.14 course . . . 225
9.21.2.15 date . . . 225
9.21.2.16 fix_quality . . . 225
9.21.2.17 satellites . . . 225
9.21.2.18 altitude . . . 225

10 File Documentation **227**

10.1 build_number.h File Reference . . . 227
10.1.1 Macro Definition Documentation . . . 227
10.1.1.1 BUILD_NUMBER . . . 227
10.2 build_number.h . . . 227
10.3 includes.h File Reference . . . 228
10.4 includes.h . . . 229
10.5 lib/clock/DS3231.cpp File Reference . . . 229
10.6 DS3231.cpp . . . 229
10.7 lib/clock/DS3231.h File Reference . . . 234
10.7.1 Macro Definition Documentation . . . 235
10.7.1.1 DS3231_DEVICE_ADRESS . . . 235
10.7.1.2 DS3231_SECONDS_REG . . . 235
10.7.1.3 DS3231_MINUTES_REG . . . 236
10.7.1.4 DS3231_HOURS_REG . . . 236
10.7.1.5 DS3231_DAY_REG . . . 236
10.7.1.6 DS3231_DATE_REG . . . 236
10.7.1.7 DS3231_MONTH_REG . . . 236
10.7.1.8 DS3231_YEAR_REG . . . 236
10.7.1.9 DS3231_CONTROL_REG . . . 237
10.7.1.10 DS3231_CONTROL_STATUS_REG . . . 237
10.7.1.11 DS3231_TEMPERATURE_MSB_REG . . . 237
10.7.1.12 DS3231_TEMPERATURE_LSB_REG . . . 237

10.7.2 Enumeration Type Documentation . . . 237
10.7.2.1 days_of_week . . . 237
10.8 DS3231.h . . . 238
10.9 lib/comms/commands/clock_commands.cpp File Reference . . . 239
10.9.1 Macro Definition Documentation . . . 240
10.9.1.1 CLOCK_GROUP . . . 240
10.9.1.2 TIME . . . 240
10.9.1.3 TIMEZONE_OFFSET . . . 240
10.9.1.4 CLOCK_SYNC_INTERVAL . . . 240
10.9.1.5 LAST_SYNC_TIME . . . 240
10.10 clock_commands.cpp . . . 241
10.11 lib/comms/commands/commands.cpp File Reference . . . 243
10.12 commands.cpp . . . 244
10.13 lib/comms/commands/commands.h File Reference . . . 245
10.14 commands.h . . . 247
10.15 lib/comms/commands/diagnostic_commands.cpp File Reference . . . 247
10.16 diagnostic_commands.cpp . . . 248
10.17 lib/comms/commands/event_commands.cpp File Reference . . . 250
10.18 event_commands.cpp . . . 250
10.19 lib/comms/commands/gps_commands.cpp File Reference . . . 251
10.19.1 Macro Definition Documentation . . . 252
10.19.1.1 GPS_GROUP . . . 252
10.19.1.2 POWER_STATUS_COMMAND . . . 252
10.19.1.3 PASSTHROUGH_COMMAND . . . 252
10.19.1.4 RMC_DATA_COMMAND . . . 252
10.19.1.5 GGA_DATA_COMMAND . . . 252
10.20 gps_commands.cpp . . . 253
10.21 lib/comms/commands/power_commands.cpp File Reference . . . 255
10.21.1 Macro Definition Documentation . . . 256
10.21.1.1 POWER_GROUP . . . 256
10.21.1.2 POWER_MANAGER_IDS . . . 256
10.21.1.3 VOLTAGE_BATTERY . . . 256
10.21.1.4 VOLTAGE_MAIN . . . 256
10.21.1.5 CHARGE_USB . . . 257
10.21.1.6 CHARGE_SOLAR . . . 257
10.21.1.7 CHARGE_TOTAL . . . 257
10.21.1.8 DRAW_TOTAL . . . 257
10.22 power_commands.cpp . . . 257
10.23 lib/comms/commands/sensor_commands.cpp File Reference . . . 259
10.23.1 Macro Definition Documentation . . . 260
10.23.1.1 SENSOR_GROUP . . . 260
10.23.1.2 SENSOR_READ . . . 260

10.23.1.3 SENSOR_CONFIGURE . . . 260
10.24 sensor_commands.cpp . . . 260
10.25 lib/comms/commands/storage_commands.cpp File Reference . . . 263
10.25.1 Macro Definition Documentation . . . 264
10.25.1.1 STORAGE_GROUP . . . 264
10.25.1.2 LIST_FILES_COMMAND . . . 264
10.25.1.3 MOUNT_COMMAND . . . 264
10.26 storage_commands.cpp . . . 265
10.27 lib/comms/commands/telemetry_commands.cpp File Reference . . . 266
10.27.1 Macro Definition Documentation . . . 267
10.27.1.1 TELEMETRY_GROUP . . . 267
10.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND . . . 267
10.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND . . . 267
10.28 telemetry_commands.cpp . . . 268
10.29 lib/comms/communication.cpp File Reference . . . 269
10.29.1 Function Documentation . . . 269
10.29.1.1 initialize_radio() . . . 269
10.29.1.2 lora_tx_done_callback() . . . 270
10.29.2 Variable Documentation . . . 271
10.29.2.1 outgoing . . . 271
10.29.2.2 msgCount . . . 271
10.29.2.3 lastSendTime . . . 271
10.29.2.4 lastReceiveTime . . . 271
10.29.2.5 lastPrintTime . . . 271
10.29.2.6 interval . . . 271
10.30 communication.cpp . . . 272
10.31 lib/comms/communication.h File Reference . . . 272
10.31.1 Function Documentation . . . 273
10.31.1.1 initialize_radio() . . . 273
10.31.1.2 lora_tx_done_callback() . . . 274
10.31.1.3 send_message() . . . 274
10.31.1.4 send_frame_uart() . . . 275
10.31.1.5 send_frame_lora() . . . 275
10.31.1.6 split_and_send_message() . . . 276
10.31.1.7 determine_unit() . . . 276
10.32 communication.h . . . 276
10.33 lib/comms/frame.cpp File Reference . . . 277
10.33.1 Detailed Description . . . 277
10.33.2 Typedef Documentation . . . 277
10.33.2.1 CommandHandler . . . 277
10.33.3 Variable Documentation . . . 278
10.33.3.1 eventRegister . . . 278

10.34 frame.cpp . . . 278
10.35 lib/comms/protocol.h File Reference . . . 280
10.35.1 Variable Documentation . . . 281
10.35.1.1 FRAME_BEGIN . . . 281
10.35.1.2 FRAME_END . . . 281
10.35.1.3 DELIMITER . . . 282
10.36 protocol.h . . . 282
10.37 lib/comms/receive.cpp File Reference . . . 283
10.37.1 Detailed Description . . . 284
10.37.2 Macro Definition Documentation . . . 284
10.37.2.1 MAX_PACKET_SIZE . . . 284
10.38 receive.cpp . . . 284
10.39 lib/comms/send.cpp File Reference . . . 285
10.39.1 Detailed Description . . . 285
10.39.2 Function Documentation . . . 286
10.39.2.1 send_message() . . . 286
10.39.2.2 send_frame_lora() . . . 286
10.39.2.3 send_frame_uart() . . . 287
10.40 send.cpp . . . 287
10.41 lib/comms/utils_converters.cpp File Reference . . . 287
10.41.1 Detailed Description . . . 288
10.42 utils_converters.cpp . . . 288
10.43 lib/eventman/event_manager.cpp File Reference . . . 289
10.43.1 Detailed Description . . . 289
10.44 event_manager.cpp . . . 290
10.45 lib/eventman/event_manager.h File Reference . . . 291
10.45.1 Detailed Description . . . 293
10.45.2 Variable Documentation . . . 293
10.45.2.1 id . . . 293
10.45.2.2 timestamp . . . 293
10.45.2.3 group . . . 293
10.45.2.4 event . . . 293
10.46 event_manager.h . . . 294
10.47 lib/location/gps_collector.cpp File Reference . . . 295
10.47.1 Detailed Description . . . 296
10.48 gps_collector.cpp . . . 296
10.49 lib/location/gps_collector.h File Reference . . . 297
10.49.1 Detailed Description . . . 298
10.50 gps_collector.h . . . 298
10.51 lib/location/NMEA/NMEA_data.h File Reference . . . 298
10.51.1 Detailed Description . . . 299
10.52 NMEA_data.h . . . 300

10.53 lib/pin_config.h File Reference . . . 301
10.53.1 Macro Definition Documentation . . . 302
10.53.1.1 DEBUG_UART_PORT . . . 302
10.53.1.2 DEBUG_UART_BAUD_RATE . . . 302
10.53.1.3 DEBUG_UART_TX_PIN . . . 302
10.53.1.4 DEBUG_UART_RX_PIN . . . 303
10.53.1.5 MAIN_I2C_PORT . . . 303
10.53.1.6 MAIN_I2C_SDA_PIN . . . 303
10.53.1.7 MAIN_I2C_SCL_PIN . . . 303
10.53.1.8 GPS_UART_PORT . . . 303
10.53.1.9 GPS_UART_BAUD_RATE . . . 303
10.53.1.10 GPS_UART_TX_PIN . . . 303
10.53.1.11 GPS_UART_RX_PIN . . . 303
10.53.1.12 GPS_POWER_ENABLE_PIN . . . 304
10.53.1.13 BUFFER_SIZE . . . 304
10.53.1.14 SD_SPI_PORT . . . 304
10.53.1.15 SD_MISO_PIN . . . 304
10.53.1.16 SD_MOSI_PIN . . . 304
10.53.1.17 SD_SCK_PIN . . . 304
10.53.1.18 SD_CS_PIN . . . 304
10.53.1.19 SD_CARD_DETECT_PIN . . . 304
10.53.1.20 SX1278_MISO . . . 305
10.53.1.21 SX1278_CS . . . 305
10.53.1.22 SX1278_SCK . . . 305
10.53.1.23 SX1278_MOSI . . . 305
10.53.1.24 SPI_PORT . . . 305
10.53.1.25 READ_BIT . . . 305
10.53.1.26 LORA_DEFAULT_SPI . . . 305
10.53.1.27 LORA_DEFAULT_SPI_FREQUENCY . . . 305
10.53.1.28 LORA_DEFAULT_SS_PIN . . . 306
10.53.1.29 LORA_DEFAULT_RESET_PIN . . . 306
10.53.1.30 LORA_DEFAULT_DIO0_PIN . . . 306
10.53.1.31 PA_OUTPUT_RFO_PIN . . . 306
10.53.1.32 PA_OUTPUT_PA_BOOST_PIN . . . 306
10.53.2 Variable Documentation . . . 306
10.53.2.1 lora_cs_pin . . . 306
10.53.2.2 lora_reset_pin . . . 306
10.53.2.3 lora_irq_pin . . . 307
10.53.2.4 lora_address_local . . . 307
10.53.2.5 lora_address_remote . . . 307
10.54 pin_config.h . . . 307
10.55 lib/powerman/INA3221/INA3221.cpp File Reference . . . 308

10.55.1 Detailed Description . . . 308
10.56 INA3221.cpp . . . 309
10.57 lib/powerman/INA3221/INA3221.h File Reference . . . 312
10.57.1 Detailed Description . . . 313
10.57.2 Enumeration Type Documentation . . . 313
10.57.2.1 ina3221_addr_t . . . 313
10.57.2.2 ina3221_ch_t . . . 314
10.57.2.3 ina3221_reg_t . . . 314
10.57.2.4 ina3221_conv_time_t . . . 315
10.57.2.5 ina3221_avg_mode_t . . . 315
10.57.3 Variable Documentation . . . 315
10.57.3.1 INA3221_CH_NUM . . . 315
10.57.3.2 SHUNT_VOLTAGE_LSB_UV . . . 316
10.58 INA3221.h . . . 316
10.59 lib/powerman/PowerManager.cpp File Reference . . . 318
10.59.1 Detailed Description . . . 318
10.60 PowerManager.cpp . . . 319
10.61 lib/powerman/PowerManager.h File Reference . . . 320
10.61.1 Detailed Description . . . 321
10.62 PowerManager.h . . . 321
10.63 lib/sensors/BH1750/BH1750.cpp File Reference . . . 322
10.64 BH1750.cpp . . . 323
10.65 lib/sensors/BH1750/BH1750.h File Reference . . . 324
10.65.1 Macro Definition Documentation . . . 325
10.65.1.1 _BH1750_DEVICE_ID . . . 325
10.65.1.2 _BH1750_MTREG_MIN . . . 325
10.65.1.3 _BH1750_MTREG_MAX . . . 325
10.65.1.4 _BH1750_DEFAULT_MTREG . . . 325
10.66 BH1750.h . . . 325
10.67 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference . . . 326
10.68 BH1750_WRAPPER.cpp . . . 326
10.69 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference . . . 327
10.70 BH1750_WRAPPER.h . . . 328
10.71 lib/sensors/BME280/BME280.cpp File Reference . . . 328
10.72 BME280.cpp . . . 329
10.73 lib/sensors/BME280/BME280.h File Reference . . . 332
10.74 BME280.h . . . 333
10.75 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference . . . 335
10.76 BME280_WRAPPER.cpp . . . 335
10.77 lib/sensors/BME280/BME280_WRAPPER.h File Reference . . . 336
10.78 BME280_WRAPPER.h . . . 336
10.79 lib/sensors/ISensor.cpp File Reference . . . 337

10.79.1 Detailed Description . . . 337
10.80 ISensor.cpp . . . 338
10.81 lib/sensors/ISensor.h File Reference . . . 338
10.81.1 Detailed Description . . . 340
10.82 ISensor.h . . . 340
10.83 lib/storage/storage.cpp File Reference . . . 341
10.83.1 Detailed Description . . . 341
10.84 storage.cpp . . . 341
10.85 lib/storage/storage.h File Reference . . . 342
10.85.1 Detailed Description . . . 343
10.86 storage.h . . . 343
10.87 lib/system_state_manager.h File Reference . . . 344
10.87.1 Detailed Description . . . 344
10.88 system_state_manager.h . . . 345
10.89 lib/telemetry/telemetry_manager.cpp File Reference . . . 346
10.89.1 Detailed Description . . . 347
10.90 telemetry_manager.cpp . . . 348
10.91 lib/telemetry/telemetry_manager.h File Reference . . . 352
10.91.1 Detailed Description . . . 354
10.92 telemetry_manager.h . . . 354
10.93 lib/utils.cpp File Reference . . . 355
10.93.1 Detailed Description . . . 356
10.93.2 Function Documentation . . . 356
10.93.2.1 get_level_color() . . . 356
10.93.2.2 get_level_prefix() . . . 357
10.93.2.3 uart_print() . . . 358
10.93.3 Variable Documentation . . . 360
10.93.3.1 uart_mutex . . . 360
10.94 utils.cpp . . . 360
10.95 lib/utils.h File Reference . . . 361
10.95.1 Detailed Description . . . 362
10.95.2 Macro Definition Documentation . . . 362
10.95.2.1 ANSI_RED . . . 362
10.95.2.2 ANSI_GREEN . . . 362
10.95.2.3 ANSI_YELLOW . . . 362
10.95.2.4 ANSI_BLUE . . . 362
10.95.2.5 ANSI_RESET . . . 362
10.95.3 Enumeration Type Documentation . . . 362
10.95.3.1 VerbosityLevel . . . 362
10.95.4 Function Documentation . . . 363
10.95.4.1 uart_print() . . . 363
10.96 utils.h . . . 364

10.97 main.cpp File Reference . . . 365
10.97.1 Macro Definition Documentation . . . 365
10.97.1.1 LOG_FILENAME . . . 365
10.97.2 Function Documentation . . . 366
10.97.2.1 core1_entry() . . . 366
10.97.2.2 init_pico_hw() . . . 366
10.97.2.3 init_modules() . . . 367
10.97.2.4 main() . . . 368
10.97.3 Variable Documentation . . . 368
10.97.3.1 buffer . . . 368
10.97.3.2 buffer_index . . . 368
10.98 main.cpp . . . 369
10.99 refactor.md File Reference . . . 371
10.100 test/comms/commands/test_clock_commands.cpp File Reference . . . 371
10.101 test_clock_commands.cpp . . . 371
10.102 test/comms/commands/test_diagnostic_commands.cpp File Reference . . . 371
10.102.1 Function Documentation . . . 372
10.102.1.1 test_handle_get_commands_list() . . . 372
10.102.1.2 test_handle_get_build_version() . . . 372
10.102.1.3 test_handle_verbosity() . . . 373
10.102.1.4 test_handle_enter_bootloader_mode() . . . 374
10.103 test_diagnostic_commands.cpp . . . 374
10.104 test/comms/commands/test_event_commands.cpp File Reference . . . 375
10.105 test_event_commands.cpp . . . 375
10.106 test/comms/commands/test_gps_commands.cpp File Reference . . . 375
10.107 test_gps_commands.cpp . . . 375
10.108 test/comms/commands/test_power_commands.cpp File Reference . . . 375
10.109 test_power_commands.cpp . . . 375
10.110 test/comms/commands/test_sensor_commands.cpp File Reference . . . 375
10.111 test_sensor_commands.cpp . . . 375
10.112 test/comms/commands/test_storage_commands.cpp File Reference . . . 376
10.113 test_storage_commands.cpp . . . 376
10.114 test/comms/commands/test_telemetry_commands.cpp File Reference . . . 376
10.115 test_telemetry_commands.cpp . . . 376
10.116 test/comms/test_comand_handlers.cpp File Reference . . . 376
10.116.1 Function Documentation . . . 377
10.116.1.1 send_frame_uart() . . . 377
10.116.1.2 send_frame_lora() . . . 377
10.116.1.3 setUp() . . . 378
10.116.1.4 tearDown() . . . 378
10.116.1.5 test_command_handler_get_operation() . . . 378
10.116.1.6 test_command_handler_set_operation() . . . 379

10.116.1.7 test_command_handler_invalid_operation() . . . 380
10.116.2 Variable Documentation . . . 380
10.116.2.1 uart_send_called . . . 380
10.116.2.2 lora_send_called . . . 380
10.116.2.3 last_frame_sent . . . 380
10.117 test_comand_handlers.cpp . . . 381
10.118 test/comms/test_converters.cpp File Reference . . . 382
10.118.1 Function Documentation . . . 382
10.118.1.1 test_operation_type_conversion() . . . 382
10.118.1.2 test_value_unit_type_conversion() . . . 383
10.118.1.3 test_exception_type_conversion() . . . 383
10.118.1.4 test_hex_string_conversion() . . . 384
10.119 test_converters.cpp . . . 384
10.120 test/comms/test_frame_build.cpp File Reference . . . 385
10.120.1 Function Documentation . . . 385
10.120.1.1 test_frame_build_val() . . . 385
10.120.1.2 test_frame_build_err() . . . 386
10.120.1.3 test_frame_build_get() . . . 386
10.120.1.4 test_frame_build_set() . . . 387
10.120.1.5 test_frame_build_res() . . . 387
10.120.1.6 test_frame_build_seq() . . . 388
10.121 test_frame_build.cpp . . . 389
10.122 test/comms/test_frame_coding.cpp File Reference . . . 390
10.122.1 Function Documentation . . . 390
10.122.1.1 test_frame_encode_basic() . . . 390
10.122.1.2 test_frame_decode_basic() . . . 391
10.122.1.3 test_frame_decode_invalid_header() . . . 391
10.123 test_frame_coding.cpp . . . 392
10.124 test/comms/test_frame_common.h File Reference . . . 392
10.124.1 Function Documentation . . . 393
10.124.1.1 create_test_frame() . . . 393
10.125 test_frame_common.h . . . 394
10.126 test/comms/test_frame_send.cpp File Reference . . . 394
10.126.1 Function Documentation . . . 394
10.126.1.1 setUp() . . . 394
10.126.1.2 tearDown() . . . 395
10.126.1.3 test_send_frame_uart() . . . 395
10.127 test_frame_send.cpp . . . 395
10.128 test/mocks/hardware_mocks.cpp File Reference . . . 396
10.128.1 Function Documentation . . . 396
10.128.1.1 mock_uart_puts() . . . 396
10.128.1.2 mock_uart_init() . . . 397

10.128.1.3 mock_spi_write_blocking() . . . 397
10.128.1.4 mock_spi_read_blocking() . . . 397
10.128.2 Variable Documentation . . . 397
10.128.2.1 mock_uart_enabled . . . 397
10.128.2.2 uart_output_buffer . . . 397
10.128.2.3 mock_spi_enabled . . . 397
10.128.2.4 spi_output_buffer . . . 398
10.129 hardware_mocks.cpp . . . 398
10.130 test/mocks/hardware_mocks.h File Reference . . . 398
10.130.1 Function Documentation . . . 400
10.130.1.1 mock_uart_puts() . . . 400
10.130.1.2 mock_uart_init() . . . 400
10.130.1.3 mock_spi_write_blocking() . . . 400
10.130.1.4 mock_spi_read_blocking() . . . 400
10.130.2 Variable Documentation . . . 400
10.130.2.1 mock_uart_enabled . . . 400
10.130.2.2 uart_output_buffer . . . 400
10.130.2.3 mock_spi_enabled . . . 401
10.130.2.4 spi_output_buffer . . . 401
10.131 hardware_mocks.h . . . 401
10.132 test/test_mocks.cpp File Reference . . . 401
10.132.1 Function Documentation . . . 402
10.132.1.1 uart_print() . . . 402
10.132.1.2 get_last_telemetry_record_csv() . . . 402
10.132.1.3 get_last_sensor_record_csv() . . . 402
10.132.1.4 test_error_code_conversion() . . . 403
10.132.1.5 test_command_handler_get_operation() . . . 403
10.132.1.6 test_command_handler_set_operation() . . . 403
10.132.1.7 test_command_handler_invalid_operation() . . . 403
10.132.2 Variable Documentation . . . 403
10.132.2.1 powerManager . . . 403
10.132.2.2 systemClock . . . 404
10.132.2.3 nmea_data . . . 404
10.132.2.4 g_pending_bootloader_reset . . . 404
10.132.2.5 g_uart_verbosity . . . 404
10.132.2.6 uart_output_buffer . . . 404
10.132.2.7 mock_uart_enabled . . . 404
10.133 test_mocks.cpp . . . 405
10.134 test/test_runner.cpp File Reference . . . 406
10.134.1 Function Documentation . . . 407
10.134.1.1 test_frame_encode_basic() . . . 407
10.134.1.2 test_frame_decode_basic() . . . 407

10.134.1.3 test_frame_decode_invalid_header() 408
10.134.1.4 test_frame_build_get() 408
10.134.1.5 test_frame_build_set() 409
10.134.1.6 test_frame_build_res() 410
10.134.1.7 test_frame_build_seq() 410
10.134.1.8 test_frame_build_val() 411
10.134.1.9 test_frame_build_err() 411
10.134.1.10 test_operation_type_conversion() 412
10.134.1.11 test_value_unit_type_conversion() 412
10.134.1.12 test_exception_type_conversion() 413
10.134.1.13 test_hex_string_conversion() 414
10.134.1.14 test_command_handler_get_operation() 414
10.134.1.15 test_command_handler_set_operation() 415
10.134.1.16 test_command_handler_invalid_operation() 416
10.134.1.17 test_handle_get_commands_list() 416
10.134.1.18 test_handle_get_build_version() 417
10.134.1.19 test_handle_verbosity() 417
10.134.1.20 test_handle_enter_bootloader_mode() 418
10.134.1.21 test_error_code_conversion() 419
10.134.1.22 main() 419
10.135 test_runner.cpp 420
10.136 todo.md File Reference 421

Index **423**

Chapter 1

refactor

gredzikk: @workspace are all singletons build like static global objects following this descriptions Printer & the↩ Printer() { static Printer printer; return printer; } This is technically a singleton though, this function can even be a static method of a class. So it guaranties to be constructed before used unlike with global static objects, that can be created in any order, making it possible to fail unconsistently when one global object uses another, quite a common scenario.

What makes it better than common way of doing singletons with creating new instance by calling new is that object destructor will be called at the end of a program. It won't happen with dynamically allocated singleton.

Another positive side is there's no way to access singleton before it gets created, even from other static methods or from subclasses. Saves you some debugging time. iny my code? are there any other things you would improve or that need addressing or fixing redundancy, inconsistency etc. recommend me a static analysis tool and other tools which can tell me whats yet to improve

Workspace: Collecting workspace informationBased on your codebase, I can see several patterns and issues related to the singleton implementation and other architectural concerns. Let me analyze the code and provide recommendations.

1.0.1 Singleton Implementation Analysis

Your codebase contains several singleton implementations with some inconsistencies:

1. **Good implementations**:
   - `SystemStateManager` uses the recommended static local object approach:
     ```
     static SystemStateManager& get_instance() {
         static SystemStateManager instance;
         return instance;
     }
     ```
   - This implementation has proper mutex protection and deleted copy constructors/operators
2. **Inconsistent implementations**:
   - `SensorWrapper` uses the same pattern but has different implementations in test mocks
3. **Work in progress**:
   - The `NMEAData` class is being converted to a singleton but currently uses a pointer-based implementation rather than the safer static local object approach.

1.0.2 Recommended Improvements

1. **Standardize singleton pattern**: Use the static local variable approach consistently:

   ```
   static ClassName& get_instance() {
       static ClassName instance;
       return instance;
   }
   ```

   This is better than pointer-based implementations because:

   - It's thread-safe in C++11 and later
   - Handles automatic destruction
   - Avoids memory leaks
   - Ensures initialization before use

2. **Convert global instances to singletons**:

   - `PowerManager` in main.cpp should follow the singleton pattern
   - `DS3231` should be converted to follow the same pattern

3. **Improve mutex usage**:

   - Use RAII for mutex locking/unlocking to prevent missed unlocks in error paths
   - Standardize between regular and recursive mutexes based on need

4. **Address inconsistencies in class interfaces**:

   - The `Print` class has some commented-out functions and inconsistent method signatures

1.0.3 Static Analysis Tools

For a C++ embedded project like yours, I recommend:

1. **Clang-Tidy**:

   - Comprehensive static analyzer that can find a wide range of issues
   - Configurable to your coding standards
   - Can be integrated with your CMake build

2. **CodeChecker**:

   - I see you already have a .codechecker directory, so you might be using it
   - It's a good wrapper around clang tools for easier integration

3. **Cppcheck**:

   - Lightweight tool good for finding memory leaks and undefined behavior
   - Works well on embedded systems code

4. **SonarQube/SonarCloud**:

   - More comprehensive analysis including code duplication, complexity metrics
   - Provides continuous code quality monitoring

5. **PVS-Studio**:

   - Commercial but powerful for finding subtle bugs
   - Has a free usage option for open source projects

1.0.4 Configuration Example

For CMake integration with clang-tidy:

```
set(CMAKE_CXX_CLANG_TIDY
  clang-tidy;
  -checks=*,-fuchsia-*,-google-*,-zircon-*,-abseil-*,-llvm-*;
  -header-filter=.*)
```

This will run clang-tidy on every file compiled through CMake.

Would you like me to provide more specific recommendations for any particular part of the codebase?

gredzikk: @workspace also are singletons really preferred here since i will ever only need one system clock, one event manager and one nmea data class, one system state manager etc. all logs etc also should be global and accessible everywhere so tell me if this static global singleton is best approach or should i pass them as objects or even some other way/

Workspace: Collecting workspace informationBased on your codebase and requirements, I'll analyze the singletons usage and offer recommendations for improvement.

1.0.5 Analysis of Current Singleton Implementations

Your codebase uses several approaches to singletons:

1. **Local static instance approach** (proper):

   ```
   static SystemStateManager& get_instance() {
       static SystemStateManager instance;
       return instance;
   }
   ```

   This is implemented correctly in `SystemStateManager` and `EventManager`.

2. **Global instance approach** (problematic):

   ```
   // In main.cpp
   PowerManager powerManager(MAIN_I2C_PORT);
   DS3231 systemClock(MAIN_I2C_PORT);
   ```

   This approach can cause initialization order problems.

3. **Mixed approaches in testing** - using different implementations for test mocks.

1.0.6 Recommendations for Singleton Implementation

Since you only need one instance of each system service (clock, event manager, etc.), the singleton pattern is indeed appropriate. The best approach for your embedded system is:

1.0.6.1 1. Standardize on Local Static Instance Pattern

```
static ClassName& get_instance() {
    static ClassName instance;
    return instance;
}
```

Benefits:

- Thread-safe in C++11+ (initialization is guaranteed thread-safe)
- Lazy initialization (created only when first used)
- No memory leaks (automatically destroyed at program end)
- Consistent initialization order (avoids the "static initialization order fiasco")

1.0.6.2 2. Add Proper Construction Requirements

For classes that need initialization parameters (like I2C bus):

```
// In header
static ClassName& get_instance();

// In implementation
ClassName& ClassName::get_instance() {
    static ClassName instance(MAIN_I2C_PORT); // Pass any required parameters
    return instance;
}
```

1.0.6.3 3. Use Static Analysis Tools

For enforcing consistent singleton usage, I recommend:

1. **Clang-Tidy** with custom checks:
   ```
   clang-tidy --checks="readability-*,performance-*,concurrency-*" *.cpp
   ```
2. **Cppcheck** for memory and thread safety issues:
   ```
   cppcheck --enable=all --suppress=missingIncludeSystem .
   ```
3. **PVS-Studio** (if available) for deeper static analysis

1.0.7 Implementation Example

Here's how to convert your `DS3231` class to the recommended singleton pattern:

```
// In DS3231.h
class DS3231 {
private:
    i2c_inst_t *i2c;
    // Other members

    // Private constructor
    DS3231(i2c_inst_t *i2c_instance);

    // Delete copy/move operations
    DS3231(const DS3231&) = delete;
    DS3231& operator=(const DS3231&) = delete;

public:
    static DS3231& get_instance();

    // Regular methods
    int set_time(const ds3231_data_t *data);
    // ...
};

// In DS3231.cpp
DS3231& DS3231::get_instance() {
    static DS3231 instance(MAIN_I2C_PORT);
    return instance;
}
```

1.0.8 Alternatives to Consider

While the singleton pattern works well for your case, consider these alternatives:

1. **Dependency Injection** - Pass service instances to objects that need them
   - More testable but more complex
2. **Service Locator Pattern** - Central registry of services

- More flexible but introduces indirection

However, given the embedded context and your specific needs, the properly-implemented singleton pattern (using the local static approach) is likely the most suitable choice.

Would you like me to provide a concrete implementation for any specific class in your codebase?

gredzikk: @workspace are there any other design patterns i can implemenet/refactor into this code to make it better quality?

Workspace: Collecting workspace informationLooking at your codebase, I can see you're already implementing several design patterns, including Singleton. However, there are additional patterns and architectural improvements that could enhance code quality, maintainability, and testability.

1.0.9 Design Patterns to Consider

1.0.9.1 1. Dependency Injection Pattern

Instead of hardcoded singletons, implement proper dependency injection:

```
// Current approach
auto& event_manager = EventManager::get_instance();
event_manager.log_event(...);

// Better approach
class TelemetryManager {
private:
    IEventManager& event_manager_;  // Interface reference

public:
    TelemetryManager(IEventManager& event_manager) : event_manager_(event_manager) {}

    void process() {
        event_manager_.log_event(...);
    }
};
```

This dramatically improves testability while maintaining the singleton architecture at the composition root.

1.0.9.2 2. Factory Method Pattern

For your sensor system, implement factories instead of direct instantiation:

```
class SensorFactory {
public:
    static std::unique_ptr<ISensor> create_sensor(SensorType type, i2c_inst_t* i2c) {
        switch(type) {
            case SensorType::BME280:
                return std::make_unique<BME280Sensor>(i2c);
            case SensorType::INA3221:
                return std::make_unique<INA3221Sensor>(i2c);
            // Other sensor types
            default:
                return nullptr;
        }
    }
};
```

1.0.9.3 3. Observer Pattern

For event handling and notifications between subsystems:

```
class SystemStateObserver {
public:
    virtual void on_state_changed(SystemState new_state) = 0;
    virtual ~SystemStateObserver() = default;
};

class SystemStateSubject {
private:
    std::vector<SystemStateObserver*> observers_;

public:
    void attach(SystemStateObserver* observer) {
        observers_.push_back(observer);
    }

    void notify(SystemState new_state) {
        for (auto observer : observers_) {
            observer->on_state_changed(new_state);
        }
    }
};
```

1.0.9.4 4. Command Pattern

For handling different communication commands:

```
class Command {
public:
    virtual std::vector<Frame> execute(const std::string& param) = 0;
    virtual ~Command() = default;
};

class GetEventsCommand : public Command {
public:
    std::vector<Frame> execute(const std::string& param) override {
        // Implementation of handle_get_events logic
    }
};

class CommandFactory {
public:
    static std::unique_ptr<Command> create_command(OperationType op_type) {
        switch(op_type) {
            case OperationType::GET:
                return std::make_unique<GetEventsCommand>();
            // Other commands
            default:
                return nullptr;
        }
    }
};
```

1.0.9.5 5. Builder Pattern

For complex object construction like Frames:

```
class FrameBuilder {
private:
    Frame frame_;

public:
    FrameBuilder& set_operation(OperationType operation) {
        frame_.operation = operation;
        return *this;
    }

    FrameBuilder& set_subsystem(uint8_t subsystem) {
        frame_.subsystem = subsystem;
        return *this;
    }

    FrameBuilder& set_command(uint8_t command) {
        frame_.command = command;
        return *this;
```

```
    }

    FrameBuilder& set_payload(const std::string& payload) {
        frame_.payload = payload;
        return *this;
    }

    Frame build() {
        return frame_;
    }
};
```

1.0.10 Architectural Improvements

1.0.10.1 1. Service Locator Pattern

Instead of global singletons, use a centralized service registry:

```
class ServiceLocator {
private:
    static std::unordered_map<std::string, std::shared_ptr<void» services_;

public:
    template<typename T>
    static void register_service(const std::string& name, std::shared_ptr<T> service) {
        services_[name] = std::static_pointer_cast<void>(service);
    }

    template<typename T>
    static std::shared_ptr<T> get_service(const std::string& name) {
        auto it = services_.find(name);
        if (it != services_.end()) {
            return std::static_pointer_cast<T>(it->second);
        }
        return nullptr;
    }
};
```

1.0.10.2 2. Repository Pattern

For data access abstraction:

```
class EventRepository {
public:
    virtual void save(const EventLog& event) = 0;
    virtual std::vector<EventLog> load(size_t count) = 0;
    virtual ~EventRepository() = default;
};

class FileSystemEventRepository : public EventRepository {
    // Implementation using file system
};

class MemoryEventRepository : public EventRepository {
    // Implementation using memory
};
```

1.0.10.3 3. RAII for Resource Management

For safer resource handling:

```
class MutexGuard {
private:
    mutex_t& mutex_;

public:
    MutexGuard(mutex_t& mutex) : mutex_(mutex) {
        mutex_enter_blocking(&mutex_);
    }

    ~MutexGuard() {
        mutex_exit(&mutex_);
    }
};
```

1.0.11 Practical Implementation Plan

1. **Start with Interface Abstraction**: Extract interfaces from concrete implementations
2. **Implement Dependency Injection**: Refactor singletons to accept dependencies
3. **Add Command Pattern**: For communications subsystem
4. **Add Observer Pattern**: For state changes and event notifications
5. **Refactor Resource Management**: Implement RAII throughout the codebase

These changes should be incremental, starting with the most critical subsystems like the `EventManager` and `PowerManager`. This approach will greatly improve testability, maintainability, and code quality while making the codebase more robust.

Chapter 2

todo

gredzikk: #file:telemetry_commands.cpp #file:telemetry_manager.cpp #file:telemetry_manager.h create python script which generates random data in correct format and saves it to buffer (without loggin to storage), and when request frame comes on uart port it sends response with last record, port should be selectable and data generation should be enable/disable to select probably with simple tkinter ui

GitHub Copilot: I'll help you create a Python script that simulates the telemetry system with a GUI interface. Here's the implementation:

```
import tkinter as tk
from tkinter import ttk
import serial
import random
import threading
import time
from datetime import datetime
import queue
import json

class TelemetryBuffer:
    def __init__(self, size=20):
        self.size = size
        self.telemetry_buffer = []
        self.sensor_buffer = []
        self.lock = threading.Lock()

    def add_record(self, telemetry_record, sensor_record):
        with self.lock:
            self.telemetry_buffer.append(telemetry_record)
            self.sensor_buffer.append(sensor_record)
            if len(self.telemetry_buffer) > self.size:
                self.telemetry_buffer.pop(0)
                self.sensor_buffer.pop(0)

    def get_last_telemetry(self):
        with self.lock:
            return self.telemetry_buffer[-1] if self.telemetry_buffer else None

    def get_last_sensor(self):
        with self.lock:
            return self.sensor_buffer[-1] if self.sensor_buffer else None

class TelemetrySimulator:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Telemetry Simulator")
        self.buffer = TelemetryBuffer()
        self.running = False
        self.serial_port = None

        self.setup_gui()
        self.setup_serial_handler()

    def setup_gui(self):
        # Port selection
        port_frame = ttk.LabelFrame(self.root, text="Serial Port Settings")
        port_frame.grid(row=0, column=0, padx=5, pady=5, sticky="ew")

        ttk.Label(port_frame, text="Port:").grid(row=0, column=0, padx=5)
```

```
        self.port_entry = ttk.Entry(port_frame, width=15)
        self.port_entry.insert(0, "COM1")
        self.port_entry.grid(row=0, column=1, padx=5)

        ttk.Label(port_frame, text="Baud:").grid(row=0, column=2, padx=5)
        self.baud_entry = ttk.Entry(port_frame, width=10)
        self.baud_entry.insert(0, "115200")
        self.baud_entry.grid(row=0, column=3, padx=5)

        # Control buttons
        control_frame = ttk.LabelFrame(self.root, text="Controls")
        control_frame.grid(row=1, column=0, padx=5, pady=5, sticky="ew")

        self.start_btn = ttk.Button(control_frame, text="Start", command=self.start_simulation)
        self.start_btn.grid(row=0, column=0, padx=5, pady=5)

        self.stop_btn = ttk.Button(control_frame, text="Stop", command=self.stop_simulation)
        self.stop_btn.grid(row=0, column=1, padx=5, pady=5)
        self.stop_btn.state(['disabled'])

        # Status display
        self.status_var = tk.StringVar(value="Stopped")
        ttk.Label(self.root, textvariable=self.status_var).grid(row=2, column=0, pady=5)

    def setup_serial_handler(self):
        self.serial_queue = queue.Queue()
        self.serial_thread = threading.Thread(target=self.handle_serial, daemon=True)
        self.serial_thread.start()

    def generate_telemetry(self):
        while self.running:
            timestamp = int(time.time())

            # Telemetry record
            telemetry = {
                "timestamp": timestamp,
                "build_version": "1.0.0",
                "battery_voltage": random.uniform(3.3, 4.2),
                "system_voltage": random.uniform(4.8, 5.2),
                "charge_current_usb": random.uniform(0, 500),
                "charge_current_solar": random.uniform(0, 300),
                "discharge_current": random.uniform(100, 400),
                "time": datetime.now().strftime("%H%M%S"),
                "latitude": f"{random.uniform(0, 90):.6f}",
                "lat_dir": random.choice(["N", "S"]),
                "longitude": f"{random.uniform(0, 180):.6f}",
                "lon_dir": random.choice(["E", "W"]),
                "speed": f"{random.uniform(0, 10):.1f}",
                "course": f"{random.uniform(0, 360):.1f}",
                "date": datetime.now().strftime("%d%m%y"),
                "fix_quality": str(random.randint(0, 2)),
                "satellites": str(random.randint(4, 12)),
                "altitude": f"{random.uniform(300, 500):.1f}"
            }

            # Sensor record
            sensor = {
                "timestamp": timestamp,
                "temperature": random.uniform(20, 30),
                "pressure": random.uniform(980, 1020),
                "humidity": random.uniform(40, 60),
                "light": random.uniform(0, 1000)
            }

            self.buffer.add_record(telemetry, sensor)
            time.sleep(2)  # Generate data every 2 seconds

    def handle_serial(self):
        while True:
            if self.serial_port:
                try:
                    if self.serial_port.in_waiting:
                        data = self.serial_port.readline().decode().strip()
                        if data.startswith("KBST"):
                            parts = data.split(";")
                            if len(parts) >= 4 and parts[3] == "8":
                                if parts[4] == "2":  # Get last telemetry record
                                    record = self.buffer.get_last_telemetry()
                                    if record:
                                        response = self.format_telemetry_csv(record)
                                        self.serial_port.write(f"KBST;0;VAL;8;2;{response};TSBK\n".encode())
                                elif parts[4] == "3":  # Get last sensor record
                                    record = self.buffer.get_last_sensor()
                                    if record:
                                        response = self.format_sensor_csv(record)
                                        self.serial_port.write(f"KBST;0;VAL;8;3;{response};TSBK\n".encode())
                time.sleep(0.1)
```

```
            else:
                time.sleep(0.5)

    def format_telemetry_csv(self, record):
        return f"{record['timestamp']},{record['build_version']},{record['battery_voltage']:.3f}," \
               f"{record['system_voltage']:.3f},{record['charge_current_usb']:.3f}," \
               f"{record['charge_current_solar']:.3f},{record['discharge_current']:.3f}," \
               f"{record['time']},{record['latitude']},{record['lat_dir']}," \
               f"{record['longitude']},{record['lon_dir']},{record['speed']}," \
               f"{record['course']},{record['date']},{record['fix_quality']}," \
               f"{record['satellites']},{record['altitude']}"

    def format_sensor_csv(self, record):
        return f"{record['timestamp']},{record['temperature']:.3f}," \
               f"{record['pressure']:.3f},{record['humidity']:.3f},{record['light']:.3f}"

    def start_simulation(self):
        try:
            port = self.port_entry.get()
            baud = int(self.baud_entry.get())
            self.serial_port = serial.Serial(port, baud, timeout=1)

            self.running = True
            self.generator_thread = threading.Thread(target=self.generate_telemetry, daemon=True)
            self.generator_thread.start()

            self.status_var.set(f"Running - Connected to {port}")
            self.start_btn.state(['disabled'])
            self.stop_btn.state(['!disabled'])
            self.port_entry.state(['disabled'])
            self.baud_entry.state(['disabled'])

        except Exception as e:
            self.status_var.set(f"Error: {str(e)}")

    def stop_simulation(self):
        self.running = False
        if self.serial_port:
            self.serial_port.close()
            self.serial_port = None

        self.status_var.set("Stopped")
        self.start_btn.state(['!disabled'])
        self.stop_btn.state(['disabled'])
        self.port_entry.state(['!disabled'])
        self.baud_entry.state(['!disabled'])

    def run(self):
        self.root.mainloop()

if __name__ == "__main__":
    simulator = TelemetrySimulator()
    simulator.run()
```

This script creates a simulator that:

1. Provides a GUI interface to:
    - Select serial port and baud rate
    - Start/stop data generation
    - Show connection status
2. Generates random telemetry data including:
    - Power system measurements
    - GPS data
    - Sensor readings
3. Maintains a circular buffer of records (max 20 entries)
4. Monitors the serial port for command frames:
    - Responds to command 8.2 with last telemetry record
    - Responds to command 8.3 with last sensor record

To use the simulator:

1. Install required dependencies:
   `pip install pyserial`
2. Run the script:
   `python telemetry_simulator.py`
3. In the GUI:
   - Enter the correct COM port for your system
   - Set the baud rate (default 115200)
   - Click Start to begin generating data
   - Click Stop to end simulation

The simulator will respond to the same command frames as the actual firmware:

- `KBST;0;GET;8;2;;TSBK` - Get last telemetry record
- `KBST;0;GET;8;3;;TSBK` - Get last sensor record

The response format matches the CSV format defined in the firmware.

gredzikk: @workspace

Chapter 3

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 4

Topic Index

4.1 Topics

Here is a list of all topics with brief descriptions:

Clock Management Commands . . . 23
Command System . . . 27
Diagnostic Commands . . . 29
Event Commands . . . 34
GPS Commands . . . 37
Power Commands . . . 41
Sensor Commands . . . 48
Storage Commands . . . 51
Telemetry Buffer Commands . . . 53
Frame Handling . . . 55
Protocol . . . 60
Receiving Data . . . 63
Utility Converters . . . 65
RTC clock . . . 71
Event Management . . . 92
Location . . . 99
INA3221 Power Monitor . . . 102
- Configuration Functions . . . 102
- Measurement Functions . . . 112

Power Management . . . 114
Sensors . . . 121
Storage . . . 126
System State Manager . . . 128
Telemetry Manager . . . 128

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

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

BH1750 145
BME280 153
BME280CalibParam 162
INA3221::conf_reg_t 169
DS3231 171
ds3231_data_t 176
EventEmitter 178
EventLog 179
EventManager 180
Frame 185
INA3221 187
ISensor 193
  BH1750Wrapper 148
  BME280Wrapper 165
INA3221::masken_reg_t 195
NMEAData 198
PowerManager 204
SensorDataRecord 209
SensorWrapper 211
SystemStateManager 213
TelemetryRecord 222

Chapter 6

Class Index

6.1 Class List

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

BH1750 145
BH1750Wrapper 148
BME280 153
BME280CalibParam 162
BME280Wrapper 165
INA3221::conf_reg_t
Configuration register bit fields 169
DS3231
Class for interfacing with the DS3231 real-time clock 171
ds3231_data_t
Structure to hold time and date information from DS3231 176
EventEmitter
Provides a simple interface for emitting events 178
EventLog
Structure for storing event log data 179
EventManager
Manages event logging and storage 180
Frame
Represents a communication frame used for data exchange 185
INA3221
INA3221 Triple-Channel Power Monitor driver class 187
ISensor
Abstract base class for sensors 193
INA3221::masken_reg_t
Mask/Enable register bit fields 195
NMEAData
Manages parsed NMEA sentences 198
PowerManager
Manages power-related functions 204
SensorDataRecord
Structure representing a single sensor data point 209
SensorWrapper
Manages a collection of sensors 211
SystemStateManager
Manages the system state of the Kubisat firmware 213
TelemetryRecord
Structure representing a single telemetry data point 222

Chapter 7

File Index

7.1 File List

Here is a list of all files with brief descriptions:

build_number.h . . . 227
includes.h . . . 228
main.cpp . . . 365
lib/pin_config.h . . . 301
lib/system_state_manager.h
Manages the system state of the Kubisat firmware . . . 344
lib/utils.cpp
Implementation of utility functions for the Kubisat firmware . . . 355
lib/utils.h
Utility functions and definitions for the Kubisat firmware . . . 361
lib/clock/DS3231.cpp . . . 229
lib/clock/DS3231.h . . . 234
lib/comms/communication.cpp . . . 269
lib/comms/communication.h . . . 272
lib/comms/frame.cpp
Implements functions for encoding, decoding, building, and processing Frames . . . 277
lib/comms/protocol.h . . . 280
lib/comms/receive.cpp
Implements functions for receiving and processing data, including LoRa and UART input . . . 283
lib/comms/send.cpp
Implements functions for sending data, including LoRa messages and Frames . . . 285
lib/comms/utils_converters.cpp
Implements utility functions for converting between different data types . . . 287
lib/comms/commands/clock_commands.cpp . . . 239
lib/comms/commands/commands.cpp . . . 243
lib/comms/commands/commands.h . . . 245
lib/comms/commands/diagnostic_commands.cpp . . . 247
lib/comms/commands/event_commands.cpp . . . 250
lib/comms/commands/gps_commands.cpp . . . 251
lib/comms/commands/power_commands.cpp . . . 255
lib/comms/commands/sensor_commands.cpp . . . 259
lib/comms/commands/storage_commands.cpp . . . 263
lib/comms/commands/telemetry_commands.cpp . . . 266
lib/eventman/event_manager.cpp
Implementation of the Event Manager and Event Emitter classes . . . 289

lib/eventman/event_manager.h
Header file for the Event Manager and Event Emitter classes . . . 291
lib/location/gps_collector.cpp
Implementation of the GPS data collector module . . . 295
lib/location/gps_collector.h
Header file for the GPS data collector module . . . 297
lib/location/NMEA/NMEA_data.h
Header file for the NMEAData class, which manages parsed NMEA sentences . . . 298
lib/powerman/PowerManager.cpp
Implementation of the PowerManager class, which manages power-related functions . . . 318
lib/powerman/PowerManager.h
Header file for the PowerManager class, which manages power-related functions . . . 320
lib/powerman/INA3221/INA3221.cpp
Implementation of the INA3221 power monitor driver . . . 308
lib/powerman/INA3221/INA3221.h
Header file for the INA3221 triple-channel power monitor driver . . . 312
lib/sensors/ISensor.cpp
Implementation of the ISensor interface and SensorWrapper class . . . 337
lib/sensors/ISensor.h
Header file for the ISensor interface and SensorWrapper class . . . 338
lib/sensors/BH1750/BH1750.cpp . . . 322
lib/sensors/BH1750/BH1750.h . . . 324
lib/sensors/BH1750/BH1750_WRAPPER.cpp . . . 326
lib/sensors/BH1750/BH1750_WRAPPER.h . . . 327
lib/sensors/BME280/BME280.cpp . . . 328
lib/sensors/BME280/BME280.h . . . 332
lib/sensors/BME280/BME280_WRAPPER.cpp . . . 335
lib/sensors/BME280/BME280_WRAPPER.h . . . 336
lib/storage/storage.cpp
Implements file system operations for the Kubisat firmware . . . 341
lib/storage/storage.h
Header file for file system operations on the Kubisat firmware . . . 342
lib/telemetry/telemetry_manager.cpp
Implementation of telemetry collection and storage functionality . . . 346
lib/telemetry/telemetry_manager.h
System telemetry collection and logging . . . 352
test/test_mocks.cpp . . . 401
test/test_runner.cpp . . . 406
test/comms/test_comand_handlers.cpp . . . 376
test/comms/test_converters.cpp . . . 382
test/comms/test_frame_build.cpp . . . 385
test/comms/test_frame_coding.cpp . . . 390
test/comms/test_frame_common.h . . . 392
test/comms/test_frame_send.cpp . . . 394
test/comms/commands/test_clock_commands.cpp . . . 371
test/comms/commands/test_diagnostic_commands.cpp . . . 371
test/comms/commands/test_event_commands.cpp . . . 375
test/comms/commands/test_gps_commands.cpp . . . 375
test/comms/commands/test_power_commands.cpp . . . 375
test/comms/commands/test_sensor_commands.cpp . . . 375
test/comms/commands/test_storage_commands.cpp . . . 376
test/comms/commands/test_telemetry_commands.cpp . . . 376
test/mocks/hardware_mocks.cpp . . . 396
test/mocks/hardware_mocks.h . . . 398

Chapter 8

Topic Documentation

8.1 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.

8.1.1 Detailed Description

Commands for managing system time and clock settings.

8.1.2 Function Documentation

8.1.2.1 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.

Here is the call graph for this function:

8.1.2.2 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.

Here is the call graph for this function:

8.1.2.3 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.

Here is the call graph for this function:

8.1.2.4 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.

Here is the call graph for this function:

8.2 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.

8.2.1 Detailed Description

Core command system implementation.

8.2.2 Typedef Documentation

8.2.2.1 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.

8.2.2.2 CommandMap

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

Map type for storing command handlers.

Definition at line 21 of file commands.cpp.

8.2.3 Function Documentation

8.2.3.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.2.4 Variable Documentation

8.2.4.1 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.

8.3 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.

8.3.1 Detailed Description

8.3.2 Function Documentation

8.3.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.3.2.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.3.2.3 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.3.2.4 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.4 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.

8.4.1 Detailed Description

Commands for accessing and managing system event logs.

8.4.2 Function Documentation

8.4.2.1 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.

Here is the call graph for this function:

8.4.2.2 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.

Here is the call graph for this function:

8.5 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.

8.5.1 Detailed Description

Commands for controlling and monitoring the GPS module.

8.5.2 Function Documentation

8.5.2.1 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.

Here is the call graph for this function:

8.5.2.2 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.

Here is the call graph for this function:

8.5.2.3 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.

Here is the call graph for this function:

8.5.2.4 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.

Here is the call graph for this function:

8.6 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 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.

8.6.1 Detailed Description

Commands for monitoring power subsystem and battery management.

8.6.2 Function Documentation

8.6.2.1 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.

Here is the call graph for this function:

8.6.2.2 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.

Here is the call graph for this function:

8.6.2.3 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.

Here is the call graph for this function:

8.6.2.4 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.

Here is the call graph for this function:

8.6.2.5 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.

Here is the call graph for this function:

8.6.2.6 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.

Here is the call graph for this function:

8.6.2.7 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.

Here is the call graph for this function:

8.7 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.

8.7.1 Detailed Description

Commands for reading and configuring sensors.

8.7.2 Function Documentation

8.7.2.1 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.

Here is the call graph for this function:

8.7.2.2 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.

Here is the call graph for this function:

8.7.2.3 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.

Here is the call graph for this function:

8.8 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.

8.8.1 Detailed Description

Commands for interacting with the SD card storage.

8.8.2 Function Documentation

8.8.2.1 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.

Here is the call graph for this function:

8.8.2.2 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.

Here is the call graph for this function:

8.9 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.

8.9.1 Detailed Description

Commands for interacting with the telemetry buffer.

8.9.2 Function Documentation

8.9.2.1 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 37 of file telemetry_commands.cpp.

Here is the call graph for this function:

8.9.2.2 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 70 of file telemetry_commands.cpp.

Here is the call graph for this function:

8.10 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.

8.10.1 Detailed Description

Functions for encoding, decoding and building communication frames.

8.10.2 Function Documentation

8.10.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.10.2.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.10.2.3 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.10.2.4 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.11 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.

8.11.1 Detailed Description

Definitions for the communication protocol used by the satellite.

8.11.2 Enumeration Type Documentation

8.11.2.1 ErrorCode

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

Standard error codes for command responses.

Enumerator

PARAM_UNNECESSARY	
PARAM_REQUIRED	
PARAM_INVALID	
INVALID_OPERATION	

Enumerator

NOT_ALLOWED	
INVALID_FORMAT	
INVALID_VALUE	
FAIL_TO_SET	
INTERNAL_FAIL_TO_READ	
UNKNOWN_ERROR	

Definition at line 53 of file protocol.h.

8.11.2.2 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.

8.11.2.3 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.

8.11.2.4 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.

8.11.2.5 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.

8.11.2.6 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.

8.12 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.

8.12.1 Detailed Description

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

8.12.2 Function Documentation

8.12.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.12.2.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.13 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.

8.13.1 Detailed Description

8.13.2 Function Documentation

8.13.2.1 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.

Here is the caller graph for this function:

8.13.2.2 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.

Here is the caller graph for this function:

8.13.2.3 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.

Here is the caller graph for this function:

8.13.2.4 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.

Here is the caller graph for this function:

8.13.2.5 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.

Here is the caller graph for this function:

8.13.2.6 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.

Here is the caller graph for this function:

8.14 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.

8.14.1 Detailed Description

Functions for interfacing with the DS3231 RTC module.

8.14.2 Function Documentation

8.14.2.1 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.

8.14.2.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.3 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.4 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.5 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°C 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.

Here is the call graph for this function:

8.14.2.6 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.7 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.8 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.9 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.

Here is the caller graph for this function:

8.14.2.10 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.11 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.

Here is the caller graph for this function:

8.14.2.12 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.13 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.

Here is the caller graph for this function:

8.14.2.14 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.15 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.16 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.

Here is the call graph for this function:

8.14.2.17 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.18 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^{2}C 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^{2}C 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.19 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.14.2.20 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.

Here is the caller graph for this function:

8.14.2.21 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.

Here is the caller graph for this function:

8.15 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__

8.15.1 Detailed Description

Classes and enums for handling system events.

8.15.2 Macro Definition Documentation

8.15.2.1 EVENT_BUFFER_SIZE

```
#define EVENT_BUFFER_SIZE 100
```

Size of the event buffer.

Definition at line 32 of file event_manager.h.

8.15.2.2 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.

8.15.2.3 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.

8.15.3 Enumeration Type Documentation

8.15.3.1 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.

8.15.3.2 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.

8.15.3.3 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.

8.15.3.4 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.

8.15.3.5 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.

8.15.3.6 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.

8.15.4 Function Documentation

8.15.4.1 __attribute__()

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

8.15.4.2 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.

Here is the call graph for this function:

8.15.4.3 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 41 of file event_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.15.4.4 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 80 of file event_manager.cpp.

8.15.4.5 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 106 of file event_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.15.5 Variable Documentation

8.15.5.1 __attribute__

```
class EventManager __attribute__
```

8.16 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.

8.16.1 Detailed Description

Classes for handling location data.

8.16.2 Macro Definition Documentation

8.16.2.1 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.

8.16.3 Function Documentation

8.16.3.1 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.

Here is the caller graph for this function:

8.16.3.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.17 INA3221 Power Monitor

Collaboration diagram for INA3221 Power Monitor:

Topics

- Configuration Functions
- Measurement Functions

8.17.1 Detailed Description

8.17.2 Configuration Functions

Collaboration diagram for 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.

8.17.2.1 Detailed Description

Functions for configuring the INA3221 device

8.17.2.2 Function Documentation

8.17.2.2.1 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.

8.17.2.2.2 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.

Here is the call graph for this function:

8.17.2.2.3 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.

Here is the call graph for this function:

8.17.2.2.4 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.17.2.2.5 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.17.2.2.6 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.

Here is the call graph for this function:

8.17.2.2.7 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.

Here is the call graph for this function:

8.17.2.2.8 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.

Here is the call graph for this function:

8.17.2.2.9 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.

Here is the call graph for this function:

8.17.2.2.10 set_shunt_measurement_enable()

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

Enable shunt voltage measurements.

Definition at line 179 of file INA3221.cpp.

Here is the call graph for this function:

8.17.2.2.11 set_shunt_measurement_disable()

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

Disable shunt voltage measurements.

Definition at line 192 of file INA3221.cpp.

Here is the call graph for this function:

8.17.2.2.12 set_bus_measurement_enable()

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

Enable bus voltage measurements.

Definition at line 205 of file INA3221.cpp.

Here is the call graph for this function:

8.17.2.2.13 set_bus_measurement_disable()

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

Disable bus voltage measurements.

Definition at line 218 of file INA3221.cpp.

Here is the call graph for this function:

8.17.2.2.14 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.

Here is the call graph for this function:

8.17.2.2.15 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.

Here is the call graph for this function:

8.17.2.2.16 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.

Here is the call graph for this function:

8.17.3 Measurement Functions

Collaboration diagram for 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.

8.17.3.1 Detailed Description

Functions for reading voltage, current and power measurements

8.17.3.2 Function Documentation

8.17.3.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.17.3.2.2 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.

Here is the call graph for this function:

8.17.3.2.3 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.

Here is the call graph for this function:

8.18 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.

8.18.1 Detailed Description

Classes for handling power-related functions.

8.18.2 Function Documentation

8.18.2.1 PowerManager()

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

Private constructor for the singleton pattern.

Initializes the INA3221 and mutex.

Definition at line 25 of file PowerManager.cpp.

8.18.2.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.18.2.3 initialize()

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

Initializes the PowerManager.

Returns

True if initialization was successful, false otherwise.

Definition at line 45 of file PowerManager.cpp.

Here is the caller graph for this function:

8.18.2.4 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.

Here is the caller graph for this function:

8.18.2.5 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.

Here is the caller graph for this function:

8.18.2.6 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.

Here is the caller graph for this function:

8.18.2.7 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.

Here is the caller graph for this function:

8.18.2.8 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.

Here is the caller graph for this function:

8.18.2.9 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.

Here is the caller graph for this function:

8.18.2.10 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.

Here is the caller graph for this function:

8.18.2.11 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.

Here is the caller graph for this function:

8.18.2.12 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.

Here is the call graph for this function:

8.18.2.13 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.

Here is the call graph for this function:

8.19 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.

8.19.1 Detailed Description

Classes for handling sensor-related functions.

8.19.2 Enumeration Type Documentation

8.19.2.1 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.

8.19.2.2 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.

8.19.3 Function Documentation

8.19.3.1 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.

8.19.3.2 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.

Here is the caller graph for this function:

8.19.3.3 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.

Here is the caller graph for this function:

8.19.3.4 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.

8.19.3.5 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.

Here is the call graph for this function:

8.19.3.6 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.

Here is the caller graph for this function:

8.20 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.

8.20.1 Detailed Description

Classes and functions for managing file system operations.

8.20.2 Function Documentation

8.20.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.20.2.2 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.

Here is the call graph for this function:

8.21 System State Manager

Classes for handling system state management.

Classes

- class SystemStateManager

  Manages the system state of the Kubisat firmware.

8.21.1 Detailed Description

Classes for handling system state management.

8.22 Telemetry Manager

Classes

- struct TelemetryRecord

  Structure representing a single telemetry data point.
- struct SensorDataRecord

  Structure representing a single sensor data point.

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

- bool telemetry_init ()

  Initialize the telemetry system.
- 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 collect_telemetry ()

  Collect telemetry data from sensors and power subsystems.
- 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.
- 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.

Variables

- static uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS

  Current sampling interval in milliseconds.
- static 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.
- size_t sensor_data_buffer_count = 0
- size_t sensor_data_buffer_write_index = 0
- mutex_t telemetry_mutex

  Mutex for thread-safe access to the telemetry buffer.
- constexpr int TELEMETRY_BUFFER_SIZE = 20

  Circular buffer for telemetry records.
- TelemetryRecord telemetry_buffer [TELEMETRY_BUFFER_SIZE]

  Circular buffer for telemetry records.
- size_t telemetry_buffer_count
- size_t telemetry_buffer_write_index
- SensorDataRecord sensor_data_buffer [TELEMETRY_BUFFER_SIZE]

  Circular buffer for sensor data records.
- size_t sensor_data_buffer_count
- size_t sensor_data_buffer_write_index
- mutex_t telemetry_mutex

  Mutex for thread-safe access to the telemetry buffer.
- const int TELEMETRY_BUFFER_SIZE

  Circular buffer for telemetry records.

8.22.1 Detailed Description

8.22.2 Macro Definition Documentation

8.22.2.1 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.

8.22.2.2 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.

8.22.2.3 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.

8.22.2.4 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.

8.22.3 Function Documentation

8.22.3.1 telemetry_init()

```
bool telemetry_init ()
```

Initialize the telemetry system.

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

Definition at line 80 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.2 collect_power_telemetry()

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

Collects power subsystem telemetry data.

Parameters

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

Definition at line 131 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.3 emit_power_events()

```
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.

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 146 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.4 collect_gps_telemetry()

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

Collects GPS telemetry data.

Parameters

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

Definition at line 190 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.5 collect_sensor_telemetry()

```
void 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 234 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.6 collect_telemetry()

```
bool 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 249 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.7 flush_telemetry()

```
bool flush_telemetry ()
```

Save buffered telemetry 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

Definition at line 291 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.8 flush_sensor_data()

```
bool 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

Definition at line 342 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.9 is_telemetry_collection_time()

```
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.

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 392 of file telemetry_manager.cpp.

Here is the caller graph for this function:

8.22.3.10 is_telemetry_flush_time()

```
bool 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 408 of file telemetry_manager.cpp.

Here is the caller graph for this function:

8.22.3.11 get_last_telemetry_record_csv()

```
std::string 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 421 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.12 get_last_sensor_record_csv()

```
std::string 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 443 of file telemetry_manager.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

8.22.3.13 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.

Here is the caller graph for this function:

8.22.3.14 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.

Here is the caller graph for this function:

8.22.4 Variable Documentation

8.22.4.1 sample_interval_ms

```
uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS [static]
```

Current sampling interval in milliseconds.

Definition at line 47 of file telemetry_manager.cpp.

8.22.4.2 flush_threshold

```
uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD  [static]
```

Current flush threshold (number of records that triggers a flush)

Definition at line 52 of file telemetry_manager.cpp.

8.22.4.3 telemetry_buffer [1/2]

```
TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE]
```

Circular buffer for telemetry records.

Definition at line 57 of file telemetry_manager.cpp.

8.22.4.4 telemetry_buffer_count [1/2]

```
size_t telemetry_buffer_count = 0
```

Definition at line 58 of file telemetry_manager.cpp.

8.22.4.5 telemetry_buffer_write_index [1/2]

```
size_t telemetry_buffer_write_index = 0
```

Definition at line 59 of file telemetry_manager.cpp.

8.22.4.6 sensor_data_buffer [1/2]

```
SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE]
```

Circular buffer for sensor data records.

Definition at line 64 of file telemetry_manager.cpp.

8.22.4.7 sensor_data_buffer_count [1/2]

```
size_t sensor_data_buffer_count = 0
```

Definition at line 65 of file telemetry_manager.cpp.

8.22.4.8 sensor_data_buffer_write_index [1/2]

```
size_t sensor_data_buffer_write_index = 0
```

Definition at line 66 of file telemetry_manager.cpp.

8.22.4.9 telemetry_mutex [1/2]

```
mutex_t telemetry_mutex
```

Mutex for thread-safe access to the telemetry buffer.

Definition at line 71 of file telemetry_manager.cpp.

8.22.4.10 TELEMETRY_BUFFER_SIZE [1/2]

```
const int TELEMETRY_BUFFER_SIZE = 20  [constexpr]
```

Circular buffer for telemetry records.

Definition at line 137 of file telemetry_manager.h.

8.22.4.11 telemetry_buffer [2/2]

```
TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE]  [extern]
```

Circular buffer for telemetry records.

Definition at line 57 of file telemetry_manager.cpp.

8.22.4.12 telemetry_buffer_count [2/2]

```
size_t telemetry_buffer_count  [extern]
```

Definition at line 58 of file telemetry_manager.cpp.

8.22.4.13 telemetry_buffer_write_index [2/2]

```
size_t telemetry_buffer_write_index  [extern]
```

Definition at line 59 of file telemetry_manager.cpp.

8.22.4.14 sensor_data_buffer [2/2]

```
SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE]  [extern]
```

Circular buffer for sensor data records.

Definition at line 64 of file telemetry_manager.cpp.

8.22.4.15 sensor_data_buffer_count [2/2]

```
size_t sensor_data_buffer_count  [extern]
```

Definition at line 65 of file telemetry_manager.cpp.

8.22.4.16 sensor_data_buffer_write_index [2/2]

```
size_t sensor_data_buffer_write_index [extern]
```

Definition at line 66 of file telemetry_manager.cpp.

8.22.4.17 telemetry_mutex [2/2]

```
mutex_t telemetry_mutex [extern]
```

Mutex for thread-safe access to the telemetry buffer.

Definition at line 71 of file telemetry_manager.cpp.

8.22.4.18 TELEMETRY_BUFFER_SIZE [2/2]

```
const int TELEMETRY_BUFFER_SIZE [extern], [constexpr]
```

Circular buffer for telemetry records.

Definition at line 138 of file telemetry_manager.h.

Chapter 9

Class Documentation

9.1 BH1750 Class Reference

```
#include <BH1750.h>
```

Public Types

- 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 }

Public Member Functions

- BH1750 (i2c_inst_t *i2c, uint8_t addr=0x23)
- bool begin (Mode mode=Mode::CONTINUOUS_HIGH_RES_MODE)
- void configure (Mode mode)
- float get_light_level ()

Private Member Functions

- void write8 (uint8_t data)

Private Attributes

- uint8_t _i2c_addr
- i2c_inst_t * i2c_port_

9.1.1 Detailed Description

Definition at line 12 of file BH1750.h.

9.1.2 Member Enumeration Documentation

9.1.2.1 Mode

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

Enumerator

UNCONFIGURED_POWER_DOWN	
POWER_ON	
RESET	
CONTINUOUS_HIGH_RES_MODE	
CONTINUOUS_HIGH_RES_MODE↩ _2	
CONTINUOUS_LOW_RES_MODE	
ONE_TIME_HIGH_RES_MODE	
ONE_TIME_HIGH_RES_MODE_2	
ONE_TIME_LOW_RES_MODE	

Definition at line 15 of file BH1750.h.

9.1.3 Constructor & Destructor Documentation

9.1.3.1 BH1750()

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

Definition at line 6 of file BH1750.cpp.

9.1.4 Member Function Documentation

9.1.4.1 begin()

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

Definition at line 8 of file BH1750.cpp.

Here is the call graph for this function:

9.1.4.2 configure()

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

Definition at line 21 of file BH1750.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

9.1.4.3 get_light_level()

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

Definition at line 39 of file BH1750.cpp.

9.1.4.4 write8()

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

Definition at line 48 of file BH1750.cpp.

Here is the caller graph for this function:

9.1.5 Member Data Documentation

9.1.5.1 _i2c_addr

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

Definition at line 34 of file BH1750.h.

9.1.5.2 i2c_port_

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

Definition at line 35 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

9.2 BH1750Wrapper Class Reference

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

Inheritance diagram for BH1750Wrapper:

Collaboration 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
- i2c_inst_t * i2c_port_

9.2.1 Detailed Description

Definition at line 9 of file BH1750_WRAPPER.h.

9.2.2 Constructor & Destructor Documentation

9.2.2.1 BH1750Wrapper() [1/2]

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

Definition at line 5 of file BH1750_WRAPPER.cpp.

9.2.2.2 BH1750Wrapper() [2/2]

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

9.2.3 Member Function Documentation

9.2.3.1 get_i2c_addr()

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

9.2.3.2 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.

Here is the caller graph for this function:

9.2.3.3 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.

9.2.3.4 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.

9.2.3.5 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.

9.2.3.6 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.

9.2.3.7 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 26 of file BH1750_WRAPPER.h.

Here is the caller graph for this function:

9.2.4 Member Data Documentation

9.2.4.1 sensor_

```
BH1750 BH1750Wrapper::sensor_ [private]
```

Definition at line 11 of file BH1750_WRAPPER.h.

9.2.4.2 initialized_

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

Definition at line 12 of file BH1750_WRAPPER.h.

9.2.4.3 i2c_port_

```
i2c_inst_t* BH1750Wrapper::i2c_port_ [private]
```

Definition at line 13 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

9.3 BME280 Class Reference

```
#include <BME280.h>
```

Collaboration diagram for BME280:

Public Member Functions

- 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

Static Public Attributes

- static constexpr uint8_t ADDR_SDO_LOW = 0x76
- static constexpr uint8_t ADDR_SDO_HIGH = 0x77

Private Member Functions

- bool configure_sensor ()
- bool get_calibration_parameters ()

Private Attributes

- i2c_inst_t * i2c_port
- uint8_t device_addr
- BME280CalibParam calib_params
- bool initialized_
- int32_t t_fine

Static Private Attributes

- static constexpr uint8_t REG_CONFIG = 0xF5
- static constexpr uint8_t REG_CTRL_MEAS = 0xF4
- static constexpr uint8_t REG_CTRL_HUM = 0xF2
- static constexpr uint8_t REG_RESET = 0xE0
- static constexpr uint8_t REG_PRESSURE_MSB = 0xF7
- static constexpr uint8_t REG_TEMPERATURE_MSB = 0xFA
- static constexpr uint8_t REG_HUMIDITY_MSB = 0xFD
- static constexpr uint8_t REG_DIG_T1_LSB = 0x88
- static constexpr uint8_t REG_DIG_T1_MSB = 0x89
- static constexpr uint8_t REG_DIG_T2_LSB = 0x8A
- static constexpr uint8_t REG_DIG_T2_MSB = 0x8B
- static constexpr uint8_t REG_DIG_T3_LSB = 0x8C
- static constexpr uint8_t REG_DIG_T3_MSB = 0x8D
- static constexpr uint8_t REG_DIG_P1_LSB = 0x8E
- static constexpr uint8_t REG_DIG_P1_MSB = 0x8F
- static constexpr uint8_t REG_DIG_P2_LSB = 0x90
- static constexpr uint8_t REG_DIG_P2_MSB = 0x91
- static constexpr uint8_t REG_DIG_P3_LSB = 0x92
- static constexpr uint8_t REG_DIG_P3_MSB = 0x93
- static constexpr uint8_t REG_DIG_P4_LSB = 0x94
- static constexpr uint8_t REG_DIG_P4_MSB = 0x95
- static constexpr uint8_t REG_DIG_P5_LSB = 0x96
- static constexpr uint8_t REG_DIG_P5_MSB = 0x97
- static constexpr uint8_t REG_DIG_P6_LSB = 0x98
- static constexpr uint8_t REG_DIG_P6_MSB = 0x99
- static constexpr uint8_t REG_DIG_P7_LSB = 0x9A
- static constexpr uint8_t REG_DIG_P7_MSB = 0x9B
- static constexpr uint8_t REG_DIG_P8_LSB = 0x9C
- static constexpr uint8_t REG_DIG_P8_MSB = 0x9D
- static constexpr uint8_t REG_DIG_P9_LSB = 0x9E
- static constexpr uint8_t REG_DIG_P9_MSB = 0x9F
- static constexpr uint8_t REG_DIG_H1 = 0xA1
- static constexpr uint8_t REG_DIG_H2 = 0xE1
- static constexpr uint8_t REG_DIG_H3 = 0xE3
- static constexpr uint8_t REG_DIG_H4 = 0xE4
- static constexpr uint8_t REG_DIG_H5 = 0xE5
- static constexpr uint8_t REG_DIG_H6 = 0xE7
- static constexpr size_t NUM_CALIB_PARAMS = 24

9.3.1 Detailed Description

Definition at line 38 of file BME280.h.

9.3.2 Constructor & Destructor Documentation

9.3.2.1 BME280()

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

Definition at line 14 of file BME280.cpp.

9.3.3 Member Function Documentation

9.3.3.1 init()

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

Definition at line 18 of file BME280.cpp.

Here is the call graph for this function:

9.3.3.2 reset()

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

Definition at line 59 of file BME280.cpp.

9.3.3.3 read_raw_all()

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

Definition at line 68 of file BME280.cpp.

9.3.3.4 convert_temperature()

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

Definition at line 101 of file BME280.cpp.

9.3.3.5 convert_pressure()

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

Definition at line 110 of file BME280.cpp.

9.3.3.6 convert_humidity()

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

Definition at line 131 of file BME280.cpp.

9.3.3.7 configure_sensor()

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

Definition at line 201 of file BME280.cpp.

Here is the caller graph for this function:

9.3.3.8 get_calibration_parameters()

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

Definition at line 143 of file BME280.cpp.

Here is the caller graph for this function:

9.3.4 Member Data Documentation

9.3.4.1 ADDR_SDO_LOW

```
uint8_t BME280::ADDR_SDO_LOW = 0x76 [static], [constexpr]
```

Definition at line 41 of file BME280.h.

9.3.4.2 ADDR_SDO_HIGH

```
uint8_t BME280::ADDR_SDO_HIGH = 0x77  [static], [constexpr]
```

Definition at line 42 of file BME280.h.

9.3.4.3 i2c_port

```
i2c_inst_t* BME280::i2c_port  [private]
```

Definition at line 69 of file BME280.h.

9.3.4.4 device_addr

```
uint8_t BME280::device_addr  [private]
```

Definition at line 70 of file BME280.h.

9.3.4.5 calib_params

```
BME280CalibParam BME280::calib_params  [private]
```

Definition at line 73 of file BME280.h.

9.3.4.6 initialized_

```
bool BME280::initialized_  [private]
```

Definition at line 76 of file BME280.h.

9.3.4.7 t_fine

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

Definition at line 79 of file BME280.h.

9.3.4.8 REG_CONFIG

```
uint8_t BME280::REG_CONFIG = 0xF5  [static], [constexpr], [private]
```

Definition at line 82 of file BME280.h.

9.3.4.9 REG_CTRL_MEAS

```
uint8_t BME280::REG_CTRL_MEAS = 0xF4  [static], [constexpr], [private]
```

Definition at line 83 of file BME280.h.

9.3.4.10 REG_CTRL_HUM

```
uint8_t BME280::REG_CTRL_HUM = 0xF2  [static], [constexpr], [private]
```

Definition at line 84 of file BME280.h.

9.3.4.11 REG_RESET

```
uint8_t BME280::REG_RESET = 0xE0  [static], [constexpr], [private]
```

Definition at line 85 of file BME280.h.

9.3.4.12 REG_PRESSURE_MSB

```
uint8_t BME280::REG_PRESSURE_MSB = 0xF7  [static], [constexpr], [private]
```

Definition at line 87 of file BME280.h.

9.3.4.13 REG_TEMPERATURE_MSB

```
uint8_t BME280::REG_TEMPERATURE_MSB = 0xFA  [static], [constexpr], [private]
```

Definition at line 88 of file BME280.h.

9.3.4.14 REG_HUMIDITY_MSB

```
uint8_t BME280::REG_HUMIDITY_MSB = 0xFD  [static], [constexpr], [private]
```

Definition at line 89 of file BME280.h.

9.3.4.15 REG_DIG_T1_LSB

```
uint8_t BME280::REG_DIG_T1_LSB = 0x88  [static], [constexpr], [private]
```

Definition at line 92 of file BME280.h.

9.3.4.16 REG_DIG_T1_MSB

```
uint8_t BME280::REG_DIG_T1_MSB = 0x89  [static], [constexpr], [private]
```

Definition at line 93 of file BME280.h.

9.3.4.17 REG_DIG_T2_LSB

```
uint8_t BME280::REG_DIG_T2_LSB = 0x8A  [static], [constexpr], [private]
```

Definition at line 94 of file BME280.h.

9.3.4.18 REG_DIG_T2_MSB

```
uint8_t BME280::REG_DIG_T2_MSB = 0x8B  [static], [constexpr], [private]
```

Definition at line 95 of file BME280.h.

9.3.4.19 REG_DIG_T3_LSB

```
uint8_t BME280::REG_DIG_T3_LSB = 0x8C  [static], [constexpr], [private]
```

Definition at line 96 of file BME280.h.

9.3.4.20 REG_DIG_T3_MSB

```
uint8_t BME280::REG_DIG_T3_MSB = 0x8D  [static], [constexpr], [private]
```

Definition at line 97 of file BME280.h.

9.3.4.21 REG_DIG_P1_LSB

```
uint8_t BME280::REG_DIG_P1_LSB = 0x8E  [static], [constexpr], [private]
```

Definition at line 99 of file BME280.h.

9.3.4.22 REG_DIG_P1_MSB

```
uint8_t BME280::REG_DIG_P1_MSB = 0x8F  [static], [constexpr], [private]
```

Definition at line 100 of file BME280.h.

9.3.4.23 REG_DIG_P2_LSB

```
uint8_t BME280::REG_DIG_P2_LSB = 0x90  [static], [constexpr], [private]
```

Definition at line 101 of file BME280.h.

9.3.4.24 REG_DIG_P2_MSB

```
uint8_t BME280::REG_DIG_P2_MSB = 0x91  [static], [constexpr], [private]
```

Definition at line 102 of file BME280.h.

9.3.4.25 REG_DIG_P3_LSB

```
uint8_t BME280::REG_DIG_P3_LSB = 0x92  [static], [constexpr], [private]
```

Definition at line 103 of file BME280.h.

9.3.4.26 REG_DIG_P3_MSB

```
uint8_t BME280::REG_DIG_P3_MSB = 0x93  [static], [constexpr], [private]
```

Definition at line 104 of file BME280.h.

9.3.4.27 REG_DIG_P4_LSB

```
uint8_t BME280::REG_DIG_P4_LSB = 0x94  [static], [constexpr], [private]
```

Definition at line 105 of file BME280.h.

9.3.4.28 REG_DIG_P4_MSB

```
uint8_t BME280::REG_DIG_P4_MSB = 0x95  [static], [constexpr], [private]
```

Definition at line 106 of file BME280.h.

9.3.4.29 REG_DIG_P5_LSB

```
uint8_t BME280::REG_DIG_P5_LSB = 0x96  [static], [constexpr], [private]
```

Definition at line 107 of file BME280.h.

9.3.4.30 REG_DIG_P5_MSB

```
uint8_t BME280::REG_DIG_P5_MSB = 0x97  [static], [constexpr], [private]
```

Definition at line 108 of file BME280.h.

9.3.4.31 REG_DIG_P6_LSB

```
uint8_t BME280::REG_DIG_P6_LSB = 0x98  [static], [constexpr], [private]
```

Definition at line 109 of file BME280.h.

9.3.4.32 REG_DIG_P6_MSB

```
uint8_t BME280::REG_DIG_P6_MSB = 0x99  [static], [constexpr], [private]
```

Definition at line 110 of file BME280.h.

9.3.4.33 REG_DIG_P7_LSB

```
uint8_t BME280::REG_DIG_P7_LSB = 0x9A  [static], [constexpr], [private]
```

Definition at line 111 of file BME280.h.

9.3.4.34 REG_DIG_P7_MSB

```
uint8_t BME280::REG_DIG_P7_MSB = 0x9B  [static], [constexpr], [private]
```

Definition at line 112 of file BME280.h.

9.3.4.35 REG_DIG_P8_LSB

```
uint8_t BME280::REG_DIG_P8_LSB = 0x9C  [static], [constexpr], [private]
```

Definition at line 113 of file BME280.h.

9.3.4.36 REG_DIG_P8_MSB

```
uint8_t BME280::REG_DIG_P8_MSB = 0x9D  [static], [constexpr], [private]
```

Definition at line 114 of file BME280.h.

9.3.4.37 REG_DIG_P9_LSB

```
uint8_t BME280::REG_DIG_P9_LSB = 0x9E  [static], [constexpr], [private]
```

Definition at line 115 of file BME280.h.

9.3.4.38 REG_DIG_P9_MSB

```
uint8_t BME280::REG_DIG_P9_MSB = 0x9F  [static], [constexpr], [private]
```

Definition at line 116 of file BME280.h.

9.3.4.39 REG_DIG_H1

```
uint8_t BME280::REG_DIG_H1 = 0xA1  [static], [constexpr], [private]
```

Definition at line 119 of file BME280.h.

9.3.4.40 REG_DIG_H2

```
uint8_t BME280::REG_DIG_H2 = 0xE1  [static], [constexpr], [private]
```

Definition at line 120 of file BME280.h.

9.3.4.41 REG_DIG_H3

```
uint8_t BME280::REG_DIG_H3 = 0xE3  [static], [constexpr], [private]
```

Definition at line 121 of file BME280.h.

9.3.4.42 REG_DIG_H4

```
uint8_t BME280::REG_DIG_H4 = 0xE4  [static], [constexpr], [private]
```

Definition at line 122 of file BME280.h.

9.3.4.43 REG_DIG_H5

```
uint8_t BME280::REG_DIG_H5 = 0xE5  [static], [constexpr], [private]
```

Definition at line 123 of file BME280.h.

9.3.4.44 REG_DIG_H6

```
uint8_t BME280::REG_DIG_H6 = 0xE7  [static], [constexpr], [private]
```

Definition at line 124 of file BME280.h.

9.3.4.45 NUM_CALIB_PARAMS

```
size_t BME280::NUM_CALIB_PARAMS = 24  [static], [constexpr], [private]
```

Definition at line 127 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

9.4 BME280CalibParam Struct Reference

```
#include <BME280.h>
```

Public Attributes

- 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

9.4.1 Detailed Description

Definition at line 11 of file BME280.h.

9.4.2 Member Data Documentation

9.4.2.1 dig_t1

```
uint16_t BME280CalibParam::dig_t1
```

Definition at line 13 of file BME280.h.

9.4.2.2 dig_t2

```
int16_t BME280CalibParam::dig_t2
```

Definition at line 14 of file BME280.h.

9.4.2.3 dig_t3

```
int16_t BME280CalibParam::dig_t3
```

Definition at line 15 of file BME280.h.

9.4.2.4 dig_p1

```
uint16_t BME280CalibParam::dig_p1
```

Definition at line 18 of file BME280.h.

9.4.2.5 dig_p2

```
int16_t BME280CalibParam::dig_p2
```

Definition at line 19 of file BME280.h.

9.4.2.6 dig_p3

```
int16_t BME280CalibParam::dig_p3
```

Definition at line 20 of file BME280.h.

9.4.2.7 dig_p4

```
int16_t BME280CalibParam::dig_p4
```

Definition at line 21 of file BME280.h.

9.4.2.8 dig_p5

```
int16_t BME280CalibParam::dig_p5
```

Definition at line 22 of file BME280.h.

9.4.2.9 dig_p6

```
int16_t BME280CalibParam::dig_p6
```

Definition at line 23 of file BME280.h.

9.4.2.10 dig_p7

```
int16_t BME280CalibParam::dig_p7
```

Definition at line 24 of file BME280.h.

9.4.2.11 dig_p8

```
int16_t BME280CalibParam::dig_p8
```

Definition at line 25 of file BME280.h.

9.4.2.12 dig_p9

```
int16_t BME280CalibParam::dig_p9
```

Definition at line 26 of file BME280.h.

9.4.2.13 dig_h1

```
uint8_t BME280CalibParam::dig_h1
```

Definition at line 29 of file BME280.h.

9.4.2.14 dig_h2

```
int16_t BME280CalibParam::dig_h2
```

Definition at line 30 of file BME280.h.

9.4.2.15 dig_h3

```
uint8_t BME280CalibParam::dig_h3
```

Definition at line 31 of file BME280.h.

9.4.2.16 dig_h4

```
int16_t BME280CalibParam::dig_h4
```

Definition at line 32 of file BME280.h.

9.4.2.17 dig_h5

```
int16_t BME280CalibParam::dig_h5
```

Definition at line 33 of file BME280.h.

9.4.2.18 dig_h6

```
int8_t BME280CalibParam::dig_h6
```

Definition at line 34 of file BME280.h.

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

- lib/sensors/BME280/BME280.h

9.5 BME280Wrapper Class Reference

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

Inheritance diagram for BME280Wrapper:

Collaboration 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

9.5.1 Detailed Description

Definition at line 8 of file BME280_WRAPPER.h.

9.5.2 Constructor & Destructor Documentation

9.5.2.1 BME280Wrapper()

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

Definition at line 3 of file BME280_WRAPPER.cpp.

9.5.3 Member Function Documentation

9.5.3.1 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.

Here is the caller graph for this function:

9.5.3.2 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.

9.5.3.3 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.

9.5.3.4 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.

9.5.3.5 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.

9.5.3.6 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.

Here is the caller graph for this function:

9.5.4 Member Data Documentation

9.5.4.1 sensor_

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

Definition at line 10 of file BME280_WRAPPER.h.

9.5.4.2 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

9.6 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

9.6.1 Detailed Description

Configuration register bit fields.

Definition at line 101 of file INA3221.h.

9.6.2 Member Data Documentation

9.6.2.1 mode_shunt_en

```
uint16_t INA3221::conf_reg_t::mode_shunt_en
```

Definition at line 102 of file INA3221.h.

9.6.2.2 mode_bus_en

```
uint16_t INA3221::conf_reg_t::mode_bus_en
```

Definition at line 103 of file INA3221.h.

9.6.2.3 mode_continious_en

```
uint16_t INA3221::conf_reg_t::mode_continious_en
```

Definition at line 104 of file INA3221.h.

9.6.2.4 shunt_conv_time

```
uint16_t INA3221::conf_reg_t::shunt_conv_time
```

Definition at line 105 of file INA3221.h.

9.6.2.5 bus_conv_time

```
uint16_t INA3221::conf_reg_t::bus_conv_time
```

Definition at line 106 of file INA3221.h.

9.6.2.6 avg_mode

```
uint16_t INA3221::conf_reg_t::avg_mode
```

Definition at line 107 of file INA3221.h.

9.6.2.7 ch3_en

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

Definition at line 108 of file INA3221.h.

9.6.2.8 ch2_en

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

Definition at line 109 of file INA3221.h.

9.6.2.9 ch1_en

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

Definition at line 110 of file INA3221.h.

9.6.2.10 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

9.7 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

9.7.1 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.

9.7.2 Constructor & Destructor Documentation

9.7.2.1 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

Here is the call graph for this function:

Here is the caller graph for this function:

9.7.2.2 DS3231() [2/2]

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

Here is the call graph for this function:

9.7.3 Member Function Documentation

9.7.3.1 operator=()

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

Here is the call graph for this function:

9.7.4 Member Data Documentation

9.7.4.1 i2c

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

Definition at line 232 of file DS3231.h.

9.7.4.2 ds3231_addr

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

Definition at line 233 of file DS3231.h.

9.7.4.3 clock_mutex_

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

Definition at line 234 of file DS3231.h.

9.7.4.4 timezone_offset_minutes_

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

Definition at line 235 of file DS3231.h.

9.7.4.5 sync_interval_minutes_

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

Definition at line 236 of file DS3231.h.

9.7.4.6 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

9.8 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)

9.8.1 Detailed Description

Structure to hold time and date information from DS3231.

Definition at line 90 of file DS3231.h.

9.8.2 Member Data Documentation

9.8.2.1 seconds

```
uint8_t ds3231_data_t::seconds
```

Seconds (0-59)

Definition at line 91 of file DS3231.h.

9.8.2.2 minutes

```
uint8_t ds3231_data_t::minutes
```

Minutes (0-59)

Definition at line 92 of file DS3231.h.

9.8.2.3 hours

```
uint8_t ds3231_data_t::hours
```

Hours (0-23)

Definition at line 93 of file DS3231.h.

9.8.2.4 day

```
uint8_t ds3231_data_t::day
```

Day of the week (1-7)

Definition at line 94 of file DS3231.h.

9.8.2.5 date

```
uint8_t ds3231_data_t::date
```

Date (1-31)

Definition at line 95 of file DS3231.h.

9.8.2.6 month

```
uint8_t ds3231_data_t::month
```

Month (1-12)

Definition at line 96 of file DS3231.h.

9.8.2.7 year

```
uint8_t ds3231_data_t::year
```

Year (0-99)

Definition at line 97 of file DS3231.h.

9.8.2.8 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

9.9 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.

9.9.1 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.

9.9.2 Member Function Documentation

9.9.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

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

- lib/eventman/event_manager.h

9.10 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.

9.10.1 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.

9.10.2 Member Data Documentation

9.10.2.1 id

```
uint16_t EventLog::id
```

Unique identifier for the event.

Definition at line 172 of file event_manager.h.

9.10.2.2 timestamp

```
uint32_t EventLog::timestamp
```

Timestamp of the event in milliseconds since boot.

Definition at line 174 of file event_manager.h.

9.10.2.3 group

```
uint8_t EventLog::group
```

Event group.

Definition at line 176 of file event_manager.h.

9.10.2.4 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

9.11 EventManager Class Reference

Manages event logging and storage.

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

Collaboration diagram for EventManager:

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

9.11.1 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.

9.11.2 Constructor & Destructor Documentation

9.11.2.1 EventManager() [1/2]

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

Definition at line 199 of file event_manager.h.

Here is the caller graph for this function:

9.11.2.2 EventManager() [2/2]

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

Here is the call graph for this function:

9.11.3 Member Function Documentation

9.11.3.1 operator=()

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

Here is the call graph for this function:

9.11.3.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

9.11.3.3 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.

9.11.3.4 load_from_storage()

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

Loads the event buffer from persistent storage.

Returns

True if the load was successful, false otherwise.

Here is the caller graph for this function:

9.11.4 Member Data Documentation

9.11.4.1 events

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

Definition at line 192 of file event_manager.h.

9.11.4.2 eventCount

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

Definition at line 193 of file event_manager.h.

9.11.4.3 writeIndex

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

Definition at line 194 of file event_manager.h.

9.11.4.4 eventMutex

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

Definition at line 195 of file event_manager.h.

9.11.4.5 nextEventId

```
uint16_t EventManager::nextEventId [private]
```

Definition at line 196 of file event_manager.h.

9.11.4.6 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

9.12 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

9.12.1 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.

9.12.2 Member Data Documentation

9.12.2.1 header

```
std::string Frame::header
```

Definition at line 182 of file protocol.h.

9.12.2.2 direction

```
uint8_t Frame::direction
```

Definition at line 183 of file protocol.h.

9.12.2.3 operationType

```
OperationType Frame::operationType
```

Definition at line 184 of file protocol.h.

9.12.2.4 group

```
uint8_t Frame::group
```

Definition at line 185 of file protocol.h.

9.12.2.5 command

```
uint8_t Frame::command
```

Definition at line 186 of file protocol.h.

9.12.2.6 value

```
std::string Frame::value
```

Definition at line 187 of file protocol.h.

9.12.2.7 unit

```
std::string Frame::unit
```

Definition at line 188 of file protocol.h.

9.12.2.8 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

9.13 INA3221 Class Reference

INA3221 Triple-Channel Power Monitor driver class.

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

Collaboration diagram for INA3221:

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

9.13.1 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.

9.13.2 Member Function Documentation

9.13.2.1 _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.

Here is the caller graph for this function:

9.13.2.2 _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.

Here is the caller graph for this function:

9.13.2.3 get_current()

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

9.13.3 Member Data Documentation

9.13.3.1 _i2c_addr

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

Definition at line 137 of file INA3221.h.

9.13.3.2 _i2c

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

Definition at line 138 of file INA3221.h.

9.13.3.3 _shuntRes

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

Definition at line 141 of file INA3221.h.

9.13.3.4 _filterRes

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

Definition at line 144 of file INA3221.h.

9.13.3.5 _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

9.14 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.

9.14.1 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.

9.14.2 Constructor & Destructor Documentation

9.14.2.1 ~ISensor()

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

Virtual destructor.

Ensures proper cleanup of derived classes.

9.14.3 Member Function Documentation

9.14.3.1 init()

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

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implemented in BH1750Wrapper, and BME280Wrapper.

9.14.3.2 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.

9.14.3.3 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.

9.14.3.4 get_type()

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

Gets the sensor type.

Returns

The sensor type.

Implemented in BH1750Wrapper, and BME280Wrapper.

9.14.3.5 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.

9.14.3.6 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

9.15 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

9.15.1 Detailed Description

Mask/Enable register bit fields.

Definition at line 117 of file INA3221.h.

9.15.2 Member Data Documentation

9.15.2.1 conv_ready

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

Definition at line 118 of file INA3221.h.

9.15.2.2 timing_ctrl_alert

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

Definition at line 119 of file INA3221.h.

9.15.2.3 pwr_valid_alert

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

Definition at line 120 of file INA3221.h.

9.15.2.4 warn_alert_ch3

```
uint16_t INA3221::masken_reg_t::warn_alert_ch3
```

Definition at line 121 of file INA3221.h.

9.15.2.5 warn_alert_ch2

```
uint16_t INA3221::masken_reg_t::warn_alert_ch2
```

Definition at line 122 of file INA3221.h.

9.15.2.6 warn_alert_ch1

```
uint16_t INA3221::masken_reg_t::warn_alert_ch1
```

Definition at line 123 of file INA3221.h.

9.15.2.7 shunt_sum_alert

```
uint16_t INA3221::masken_reg_t::shunt_sum_alert
```

Definition at line 124 of file INA3221.h.

9.15.2.8 crit_alert_ch3

```
uint16_t INA3221::masken_reg_t::crit_alert_ch3
```

Definition at line 125 of file INA3221.h.

9.15.2.9 crit_alert_ch2

```
uint16_t INA3221::masken_reg_t::crit_alert_ch2
```

Definition at line 126 of file INA3221.h.

9.15.2.10 crit_alert_ch1

```
uint16_t INA3221::masken_reg_t::crit_alert_ch1
```

Definition at line 127 of file INA3221.h.

9.15.2.11 crit_alert_latch_en

```
uint16_t INA3221::masken_reg_t::crit_alert_latch_en
```

Definition at line 128 of file INA3221.h.

9.15.2.12 warn_alert_latch_en

```
uint16_t INA3221::masken_reg_t::warn_alert_latch_en
```

Definition at line 129 of file INA3221.h.

9.15.2.13 shunt_sum_en_ch3

```
uint16_t INA3221::masken_reg_t::shunt_sum_en_ch3
```

Definition at line 130 of file INA3221.h.

9.15.2.14 shunt_sum_en_ch2

```
uint16_t INA3221::masken_reg_t::shunt_sum_en_ch2
```

Definition at line 131 of file INA3221.h.

9.15.2.15 shunt_sum_en_ch1

```
uint16_t INA3221::masken_reg_t::shunt_sum_en_ch1
```

Definition at line 132 of file INA3221.h.

9.15.2.16 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

9.16 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.

9.16.1 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.

9.16.2 Constructor & Destructor Documentation

9.16.2.1 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.

Here is the caller graph for this function:

9.16.2.2 NMEAData() [2/2]

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

Deleted copy constructor to prevent copying.

Here is the call graph for this function:

9.16.3 Member Function Documentation

9.16.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

Here is the call graph for this function:

9.16.3.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

9.16.3.3 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.

Here is the caller graph for this function:

9.16.3.4 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.

Here is the caller graph for this function:

9.16.3.5 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.

9.16.3.6 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.

9.16.3.7 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.

Here is the caller graph for this function:

9.16.3.8 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.

Here is the call graph for this function:

9.16.4 Member Data Documentation

9.16.4.1 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.

9.16.4.2 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.

9.16.4.3 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.

9.16.4.4 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 files:

- lib/location/NMEA/NMEA_data.h
- test/test_mocks.cpp

9.17 PowerManager Class Reference

Manages power-related functions.

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

Collaboration diagram for PowerManager:

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.

9.17.1 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.

9.17.2 Constructor & Destructor Documentation

9.17.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

9.17.2.2 PowerManager() [2/2]

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

Deleted copy constructor to prevent copying.

Here is the call graph for this function:

9.17.3 Member Function Documentation

9.17.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

Here is the call graph for this function:

9.17.4 Member Data Documentation

9.17.4.1 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.

9.17.4.2 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.

9.17.4.3 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.

9.17.4.4 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.

9.17.4.5 ina3221_

```
INA3221 PowerManager::ina3221_  [private]
```

INA3221 instance for power monitoring.

Definition at line 124 of file PowerManager.h.

9.17.4.6 initialized_

```
bool PowerManager::initialized_  [private]
```

Flag indicating if the PowerManager is initialized.

Definition at line 126 of file PowerManager.h.

9.17.4.7 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.

9.17.4.8 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.

9.17.4.9 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

9.18 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

9.18.1 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.

9.18.2 Member Data Documentation

9.18.2.1 timestamp

```
uint32_t SensorDataRecord::timestamp
```

Unix timestamp of the record

Definition at line 111 of file telemetry_manager.h.

9.18.2.2 temperature

```
float SensorDataRecord::temperature
```

Temperature in degrees Celsius

Definition at line 112 of file telemetry_manager.h.

9.18.2.3 pressure

```
float SensorDataRecord::pressure
```

Pressure in hPa

Definition at line 113 of file telemetry_manager.h.

9.18.2.4 humidity

```
float SensorDataRecord::humidity
```

Relative humidity in %

Definition at line 114 of file telemetry_manager.h.

9.18.2.5 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

9.19 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.

9.19.1 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.

9.19.2 Constructor & Destructor Documentation

9.19.2.1 SensorWrapper()

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

Private constructor for the singleton pattern.

Definition at line 98 of file test_mocks.cpp.

Here is the caller graph for this function:

9.19.3 Member Function Documentation

9.19.3.1 get_instance()

```
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.

Here is the call graph for this function:

Here is the caller graph for this function:

9.19.4 Member Data Documentation

9.19.4.1 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
- test/test_mocks.cpp

9.20 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.

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.
- recursive_mutex_t mutex_

  Mutex for thread-safe access to the system state.

9.20.1 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.

9.20.2 Constructor & Destructor Documentation

9.20.2.1 SystemStateManager() [1/2]

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

Private constructor for the singleton pattern.

Initializes the system state and mutex.

Definition at line 49 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.2.2 SystemStateManager() [2/2]

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

Deleted copy constructor to prevent copying.

Here is the call graph for this function:

9.20.3 Member Function Documentation

9.20.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

Here is the call graph for this function:

9.20.3.2 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 72 of file system_state_manager.h.

Here is the call graph for this function:

Here is the caller graph for this function:

9.20.3.3 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 81 of file system_state_manager.h.

9.20.3.4 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 92 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.3.5 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 102 of file system_state_manager.h.

9.20.3.6 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 113 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.3.7 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 123 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.3.8 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 134 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.3.9 get_uart_verbosity()

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

Gets the UART verbosity level.

Returns

The UART verbosity level.

Definition at line 144 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.3.10 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 155 of file system_state_manager.h.

Here is the caller graph for this function:

9.20.4 Member Data Documentation

9.20.4.1 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.

9.20.4.2 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.

9.20.4.3 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.

9.20.4.4 uart_verbosity

```
VerbosityLevel SystemStateManager::uart_verbosity [private]
```

The UART verbosity level.

Definition at line 41 of file system_state_manager.h.

9.20.4.5 mutex_

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

Mutex for thread-safe access to the system state.

Definition at line 43 of file system_state_manager.h.

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

- lib/system_state_manager.h

9.21 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

9.21.1 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.

9.21.2 Member Data Documentation

9.21.2.1 timestamp

```
uint32_t TelemetryRecord::timestamp
```

Unix timestamp of the record

Definition at line 44 of file telemetry_manager.h.

9.21.2.2 build_version

```
std::string TelemetryRecord::build_version
```

Build version of the firmware

Definition at line 46 of file telemetry_manager.h.

9.21.2.3 battery_voltage

```
float TelemetryRecord::battery_voltage
```

Battery voltage in volts

Definition at line 49 of file telemetry_manager.h.

9.21.2.4 system_voltage

```
float TelemetryRecord::system_voltage
```

System 5V rail voltage in volts

Definition at line 50 of file telemetry_manager.h.

9.21.2.5 charge_current_usb

```
float TelemetryRecord::charge_current_usb
```

USB charging current in mA

Definition at line 51 of file telemetry_manager.h.

9.21.2.6 charge_current_solar

```
float TelemetryRecord::charge_current_solar
```

Solar charging current in mA

Definition at line 52 of file telemetry_manager.h.

9.21.2.7 discharge_current

```
float TelemetryRecord::discharge_current
```

Battery discharge current in mA

Definition at line 53 of file telemetry_manager.h.

9.21.2.8 time

```
std::string TelemetryRecord::time
```

UTC time from GPS

Definition at line 56 of file telemetry_manager.h.

9.21.2.9 latitude

```
std::string TelemetryRecord::latitude
```

Latitude from GPS

Definition at line 57 of file telemetry_manager.h.

9.21.2.10 lat_dir

```
std::string TelemetryRecord::lat_dir
```

N/S latitude direction

Definition at line 58 of file telemetry_manager.h.

9.21.2.11 longitude

```
std::string TelemetryRecord::longitude
```

Longitude from GPS

Definition at line 59 of file telemetry_manager.h.

9.21.2.12 lon_dir

```
std::string TelemetryRecord::lon_dir
```

E/W longitude direction

Definition at line 60 of file telemetry_manager.h.

9.21.2.13 speed

```
std::string TelemetryRecord::speed
```

Speed in knots

Definition at line 61 of file telemetry_manager.h.

9.21.2.14 course

```
std::string TelemetryRecord::course
```

Course in degrees

Definition at line 62 of file telemetry_manager.h.

9.21.2.15 date

```
std::string TelemetryRecord::date
```

Date from GPS

Definition at line 63 of file telemetry_manager.h.

9.21.2.16 fix_quality

```
std::string TelemetryRecord::fix_quality
```

GPS fix quality

Definition at line 66 of file telemetry_manager.h.

9.21.2.17 satellites

```
std::string TelemetryRecord::satellites
```

Number of satellites in view

Definition at line 67 of file telemetry_manager.h.

9.21.2.18 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 10

File Documentation

10.1 build_number.h File Reference

This graph shows which files directly or indirectly include this file:

Macros

- #define BUILD_NUMBER 478

10.1.1 Macro Definition Documentation

10.1.1.1 BUILD_NUMBER

```
#define BUILD_NUMBER 478
```

Definition at line 6 of file build_number.h.

10.2 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 478
00007
00008 #endif
```

10.3 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"
```

Include dependency graph for includes.h:

This graph shows which files directly or indirectly include this file:

10.4 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
```

10.5 lib/clock/DS3231.cpp File Reference

```
#include "DS3231.h"
#include "utils.h"
#include <stdio.h>
#include <mutex>
#include "event_manager.h"
#include "NMEA_data.h"
```

Include dependency graph for DS3231.cpp:

10.6 DS3231.cpp

Go to the documentation of this file.

```
#include "DS3231.h"
#include "utils.h"
#include <stdio.h>
#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
```

10.7 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"
```

Include dependency graph for DS3231.h:

This graph shows which files directly or indirectly include this file:

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.

10.7.1 Macro Definition Documentation

10.7.1.1 DS3231_DEVICE_ADRESS

```
#define DS3231_DEVICE_ADRESS 0x68
```

DS3231 I2C device address.

Definition at line 15 of file DS3231.h.

10.7.1.2 DS3231_SECONDS_REG

```
#define DS3231_SECONDS_REG 0x00
```

Register address: Seconds (0-59)

Definition at line 20 of file DS3231.h.

10.7.1.3 DS3231_MINUTES_REG

```
#define DS3231_MINUTES_REG 0x01
```

Register address: Minutes (0-59)

Definition at line 25 of file DS3231.h.

10.7.1.4 DS3231_HOURS_REG

```
#define DS3231_HOURS_REG 0x02
```

Register address: Hours (0-23 in 24hr mode)

Definition at line 30 of file DS3231.h.

10.7.1.5 DS3231_DAY_REG

```
#define DS3231_DAY_REG 0x03
```

Register address: Day of the week (1-7)

Definition at line 35 of file DS3231.h.

10.7.1.6 DS3231_DATE_REG

```
#define DS3231_DATE_REG 0x04
```

Register address: Date (1-31)

Definition at line 40 of file DS3231.h.

10.7.1.7 DS3231_MONTH_REG

```
#define DS3231_MONTH_REG 0x05
```

Register address: Month (1-12) & Century bit.

Definition at line 45 of file DS3231.h.

10.7.1.8 DS3231_YEAR_REG

```
#define DS3231_YEAR_REG 0x06
```

Register address: Year (00-99)

Definition at line 50 of file DS3231.h.

10.7.1.9 DS3231_CONTROL_REG

```
#define DS3231_CONTROL_REG 0x0E
```

Register address: Control register.

Definition at line 55 of file DS3231.h.

10.7.1.10 DS3231_CONTROL_STATUS_REG

```
#define DS3231_CONTROL_STATUS_REG 0x0F
```

Register address: Control/Status register.

Definition at line 60 of file DS3231.h.

10.7.1.11 DS3231_TEMPERATURE_MSB_REG

```
#define DS3231_TEMPERATURE_MSB_REG 0x11
```

Register address: Temperature register (MSB)

Definition at line 65 of file DS3231.h.

10.7.1.12 DS3231_TEMPERATURE_LSB_REG

```
#define DS3231_TEMPERATURE_LSB_REG 0x12
```

Register address: Temperature register (LSB)

Definition at line 70 of file DS3231.h.

10.7.2 Enumeration Type Documentation

10.7.2.1 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.

10.8 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
```

10.9 lib/comms/commands/clock_commands.cpp File Reference

```
#include "communication.h"
#include <time.h>
#include "DS3231.h"
```

Include dependency graph for clock_commands.cpp:

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.

10.9.1 Macro Definition Documentation

10.9.1.1 CLOCK_GROUP

```
#define CLOCK_GROUP 3
```

Definition at line 5 of file clock_commands.cpp.

10.9.1.2 TIME

```
#define TIME 0
```

Definition at line 6 of file clock_commands.cpp.

10.9.1.3 TIMEZONE_OFFSET

```
#define TIMEZONE_OFFSET 1
```

Definition at line 7 of file clock_commands.cpp.

10.9.1.4 CLOCK_SYNC_INTERVAL

```
#define CLOCK_SYNC_INTERVAL 2
```

Definition at line 8 of file clock_commands.cpp.

10.9.1.5 LAST_SYNC_TIME

```
#define LAST_SYNC_TIME 3
```

Definition at line 9 of file clock_commands.cpp.

10.10 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
```

10.11 lib/comms/commands/commands.cpp File Reference

```
#include "commands.h"
#include "communication.h"
```

Include dependency graph for commands.cpp:

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.

10.12 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
```

10.13 lib/comms/commands/commands.h File Reference

```
#include <string>
#include <functional>
#include <map>
#include "protocol.h"
```

Include dependency graph for commands.h:

This graph shows which files directly or indirectly include this file:

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 operation↩ Type)

  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 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.
- 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 operation↩ Type)

  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.

10.14 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
```

10.15 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"
```

Include dependency graph for diagnostic_commands.cpp:

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.

10.16 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;
}
```

10.17 lib/comms/commands/event_commands.cpp File Reference

```
#include "communication.h"
#include "event_manager.h"
#include <sstream>
```

Include dependency graph for event_commands.cpp:

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.

10.18 event_commands.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002 #include "event_manager.h"
00003 #include <sstream>
00004
00005
00011
00033 std::vector<Frame> handle_get_last_events(const std::string& param, OperationType operationType) {
00034     std::vector<Frame> frames;
00035     std::string error_msg;
00036
00037     if (operationType != OperationType::GET) {
00038         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00039         frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
00040         return frames;
00041     }
00042
00043     size_t count = 10; // Default number of events to return
00044     if (!param.empty()) {
00045         try {
00046             count = std::stoul(param);
00047             if (count > EVENT_BUFFER_SIZE) {
00048                 error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00049                 frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
00050                 return frames;
00051             }
00052         } catch (...) {
00053             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00054             frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
00055             return frames;
00056         }
00057     }
00058
00059     auto& event_manager = EventManager::get_instance();
00060     size_t available = event_manager.get_event_count();
00061     size_t to_return = (count == 0) ? available : std::min(count, available);
00062     size_t event_index = available;
00063
00064     while (to_return > 0) {
00065         std::stringstream ss;
00066         ss « std::hex « std::uppercase « std::setfill('0');
00067         size_t events_in_frame = 0;
00068
```

```
        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
```

10.19 lib/comms/commands/gps_commands.cpp File Reference

```
#include "communication.h"
#include "lib/location/gps_collector.h"
#include <sstream>
#include "system_state_manager.h"
```

Include dependency graph for gps_commands.cpp:

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.

10.19.1 Macro Definition Documentation

10.19.1.1 GPS_GROUP

```
#define GPS_GROUP 7
```

Definition at line 6 of file gps_commands.cpp.

10.19.1.2 POWER_STATUS_COMMAND

```
#define POWER_STATUS_COMMAND 1
```

Definition at line 7 of file gps_commands.cpp.

10.19.1.3 PASSTHROUGH_COMMAND

```
#define PASSTHROUGH_COMMAND 2
```

Definition at line 8 of file gps_commands.cpp.

10.19.1.4 RMC_DATA_COMMAND

```
#define RMC_DATA_COMMAND 3
```

Definition at line 9 of file gps_commands.cpp.

10.19.1.5 GGA_DATA_COMMAND

```
#define GGA_DATA_COMMAND 4
```

Definition at line 10 of file gps_commands.cpp.

10.20 gps_commands.cpp

Go to the documentation of this file.

```
#include "communication.h"
#include "lib/location/gps_collector.h"
#include <sstream>
#include "system_state_manager.h"

#define GPS_GROUP 7
#define POWER_STATUS_COMMAND 1
#define PASSTHROUGH_COMMAND 2
#define RMC_DATA_COMMAND 3
#define GGA_DATA_COMMAND 4


std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_str;

    if (operationType == OperationType::SET) {
        if (param.empty()) {
            error_str = error_code_to_string(ErrorCode::PARAM_REQUIRED);
            frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
            return frames;
        }

        try {
            int power_status = std::stoi(param);
            if (power_status != 0 && power_status != 1) {
                error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
                frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
                return frames;
            }
            gpio_put(GPS_POWER_ENABLE_PIN, power_status);
            EventEmitter::emit(EventGroup::GPS, power_status ? GPSEvent::POWER_ON : GPSEvent::POWER_OFF);
            frames.push_back(frame_build(OperationType::RES, GPS_GROUP, POWER_STATUS_COMMAND, std::to_string(power_status)));
            return frames;
        } catch (...) {
            error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
            frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
            return frames;
        }
    }

    // GET operation
    if (!param.empty()) {
        error_str = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
        return frames;
    }

    bool power_status = gpio_get(GPS_POWER_ENABLE_PIN);
    frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, POWER_STATUS_COMMAND, std::to_string(power_status)));
    return frames;
}


std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType operationType) {
    std::vector<Frame> frames;
    std::string error_str;

    if (!(operationType == OperationType::SET)) {
        error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
        frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
        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
```

10.21 lib/comms/commands/power_commands.cpp File Reference

`#include "communication.h"`

Include dependency graph for power_commands.cpp:

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.

10.21.1 Macro Definition Documentation

10.21.1.1 POWER_GROUP

```
#define POWER_GROUP 2
```

Definition at line 3 of file power_commands.cpp.

10.21.1.2 POWER_MANAGER_IDS

```
#define POWER_MANAGER_IDS 0
```

Definition at line 4 of file power_commands.cpp.

10.21.1.3 VOLTAGE_BATTERY

```
#define VOLTAGE_BATTERY 2
```

Definition at line 5 of file power_commands.cpp.

10.21.1.4 VOLTAGE_MAIN

```
#define VOLTAGE_MAIN 3
```

Definition at line 6 of file power_commands.cpp.

10.21.1.5 CHARGE_USB

```
#define CHARGE_USB 4
```

Definition at line 7 of file power_commands.cpp.

10.21.1.6 CHARGE_SOLAR

```
#define CHARGE_SOLAR 5
```

Definition at line 8 of file power_commands.cpp.

10.21.1.7 CHARGE_TOTAL

```
#define CHARGE_TOTAL 6
```

Definition at line 9 of file power_commands.cpp.

10.21.1.8 DRAW_TOTAL

```
#define DRAW_TOTAL 7
```

Definition at line 10 of file power_commands.cpp.

10.22 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
```

10.23 lib/comms/commands/sensor_commands.cpp File Reference

```
#include "communication.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <sstream>
#include "commands.h"
```

Include dependency graph for sensor_commands.cpp:

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.

10.23.1 Macro Definition Documentation

10.23.1.1 SENSOR_GROUP

```
#define SENSOR_GROUP 4
```

Definition at line 8 of file sensor_commands.cpp.

10.23.1.2 SENSOR_READ

```
#define SENSOR_READ 0
```

Definition at line 9 of file sensor_commands.cpp.

10.23.1.3 SENSOR_CONFIGURE

```
#define SENSOR_CONFIGURE 1
```

Definition at line 10 of file sensor_commands.cpp.

10.24 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));
```

```
        return frames;
    }

    // Parse sensor type and data type from param
    std::string sensor_type_str;
    std::string data_type_str;

    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;
}
```

10.25 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"
```

Include dependency graph for storage_commands.cpp:

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.

10.25.1 Macro Definition Documentation

10.25.1.1 STORAGE_GROUP

```
#define STORAGE_GROUP 6
```

Definition at line 10 of file storage_commands.cpp.

10.25.1.2 LIST_FILES_COMMAND

```
#define LIST_FILES_COMMAND 0
```

Definition at line 12 of file storage_commands.cpp.

10.25.1.3 MOUNT_COMMAND

```
#define MOUNT_COMMAND 4
```

Definition at line 13 of file storage_commands.cpp.

10.26 storage_commands.cpp

Go to the documentation of this file.

```
#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"

#define STORAGE_GROUP 6

#define LIST_FILES_COMMAND 0
#define MOUNT_COMMAND 4

std::vector<Frame> handle_list_files([[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, STORAGE_GROUP, LIST_FILES_COMMAND, error_msg));
        return frames;
    }

    DIR* dir;
    struct dirent* ent;
    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
```

10.27 lib/comms/commands/telemetry_commands.cpp File Reference

```
#include "commands.h"
#include "communication.h"
#include "telemetry_manager.h"
```

Include dependency graph for telemetry_commands.cpp:

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 operation↩Type)

  Handles the get last sensor record command.

Variables

- mutex_t telemetry_mutex

  Mutex for thread-safe access to the telemetry buffer.
- size_t telemetry_buffer_count
- size_t telemetry_buffer_write_index
- const int TELEMETRY_BUFFER_SIZE

  Circular buffer for telemetry records.

10.27.1 Macro Definition Documentation

10.27.1.1 TELEMETRY_GROUP

```
#define TELEMETRY_GROUP 8
```

Definition at line 5 of file telemetry_commands.cpp.

10.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND

```
#define GET_LAST_TELEMETRY_RECORD_COMMAND 2
```

Definition at line 6 of file telemetry_commands.cpp.

10.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND

```
#define GET_LAST_SENSOR_RECORD_COMMAND 3
```

Definition at line 7 of file telemetry_commands.cpp.

10.28 telemetry_commands.cpp

Go to the documentation of this file.

```
#include "commands.h"
#include "communication.h"
#include "telemetry_manager.h"

#define TELEMETRY_GROUP 8
#define GET_LAST_TELEMETRY_RECORD_COMMAND 2
#define GET_LAST_SENSOR_RECORD_COMMAND 3

extern mutex_t telemetry_mutex;
extern size_t telemetry_buffer_count;
extern size_t telemetry_buffer_write_index;
extern const int TELEMETRY_BUFFER_SIZE;


std::vector<Frame> handle_get_last_telemetry_record([[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, TELEMETRY_GROUP, GET_LAST_TELEMETRY_RECORD_COMMAND, error_msg));
        return frames;
    }

    std::string csv_data = get_last_telemetry_record_csv();

    if (csv_data.empty()) {
        error_msg = "NO_DATA";
        frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
        return frames;
    }
    frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP, GET_LAST_TELEMETRY_RECORD_COMMAND, csv_data));

    return frames;
}


std::vector<Frame> handle_get_last_sensor_record([[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, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
        return frames;
    }

    std::string csv_data = get_last_sensor_record_csv();

    if (csv_data.empty()) {
        error_msg = "NO_DATA";
        frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
        return frames;
    }

    frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND, csv_data));

    return frames;
} // TelemetryBufferCommands
```

10.29 lib/comms/communication.cpp File Reference

`#include "communication.h"`
Include dependency graph for communication.cpp:

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

10.29.1 Function Documentation

10.29.1.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.29.1.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.29.2 Variable Documentation

10.29.2.1 outgoing

```
string outgoing
```

Definition at line 3 of file communication.cpp.

10.29.2.2 msgCount

```
uint8_t msgCount = 0
```

Definition at line 4 of file communication.cpp.

10.29.2.3 lastSendTime

```
long lastSendTime = 0
```

Definition at line 5 of file communication.cpp.

10.29.2.4 lastReceiveTime

```
long lastReceiveTime = 0
```

Definition at line 6 of file communication.cpp.

10.29.2.5 lastPrintTime

```
long lastPrintTime = 0
```

Definition at line 7 of file communication.cpp.

10.29.2.6 interval

```
unsigned long interval = 0
```

Definition at line 8 of file communication.cpp.

10.30 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);
}
```

10.31 lib/comms/communication.h File Reference

```
#include <string>
#include <vector>
#include "protocol.h"
#include "event_manager.h"
```

Include dependency graph for communication.h:

This graph shows which files directly or indirectly include this file:

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)

10.31.1 Function Documentation

10.31.1.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.31.1.2 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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.31.1.3 send_message()

```
void send_message (
            std::string outgoing)
```

10.31.1.4 send_frame_uart()

```
void send_frame_uart (
            const Frame & frame)
```

Definition at line 47 of file send.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.31.1.5 send_frame_lora()

```
void send_frame_lora (
            const Frame & frame)
```

Definition at line 39 of file send.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.31.1.6 split_and_send_message()

```
void split_and_send_message (
            const uint8_t * data,
            size_t length)
```

10.31.1.7 determine_unit()

```
std::string determine_unit (
            uint8_t group,
            uint8_t command)
```

10.32 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
```

10.33 lib/comms/frame.cpp File Reference

Implements functions for encoding, decoding, building, and processing Frames.

```
#include "communication.h"
```

Include dependency graph for frame.cpp:

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

10.33.1 Detailed Description

Implements functions for encoding, decoding, building, and processing Frames.

Definition in file frame.cpp.

10.33.2 Typedef Documentation

10.33.2.1 CommandHandler

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

Definition at line 3 of file frame.cpp.

10.33.3 Variable Documentation

10.33.3.1 eventRegister

```
volatile uint16_t eventRegister  [extern]
```

10.34 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
```

10.35 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"
```

Include dependency graph for protocol.h:

This graph shows which files directly or indirectly include this file:

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 = ';'

10.35.1 Variable Documentation

10.35.1.1 FRAME_BEGIN

```
const std::string FRAME_BEGIN = "KBST"
```

Definition at line 31 of file protocol.h.

10.35.1.2 FRAME_END

```
const std::string FRAME_END = "TSBK"
```

Definition at line 38 of file protocol.h.

10.35.1.3 DELIMITER

```
const char DELIMITER = ';'
```

Definition at line 45 of file protocol.h.

10.36 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"
00020 #include "LoRa/LoRa-RP2040.h"
00021
00031 const std::string FRAME_BEGIN = "KBST";
00032
00038 const std::string FRAME_END = "TSBK";
00039
00045 const char DELIMITER = ';';
00046
00047
00053 enum class ErrorCode {
00054     PARAM_UNNECESSARY,     // Parameter provided but not needed
00055     PARAM_REQUIRED,        // Required parameter missing
00056     PARAM_INVALID,         // Parameter has invalid format or value
00057     INVALID_OPERATION,     // Operation not allowed for this command
00058     NOT_ALLOWED,           // Operation not permitted
00059     INVALID_FORMAT,        // Input format is incorrect
00060     INVALID_VALUE,         // Value is outside expected range
00061     FAIL_TO_SET,           // Failed to set provided value
00062     INTERNAL_FAIL_TO_READ,// Failed to read from device in remote
00063     UNKNOWN_ERROR          // Generic error
00064 };
00065
00066
00072 enum class OperationType {
00074     GET,
00076     SET,
00078     RES,
00080     VAL,
00082     SEQ,
00084     ERR,
00085
00086 };
00087
00088
00089
00095 enum class CommandAccessLevel {
00097     NONE,
00099     READ_ONLY,
00101     WRITE_ONLY,
00103     READ_WRITE
00104 };
00105
00106
00107
00113 enum class ValueUnit {
00115     UNDEFINED,
00117     SECOND,
00119     VOLT,
00121     BOOL,
00123     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
```

10.37 lib/comms/receive.cpp File Reference

Implements functions for receiving and processing data, including LoRa and UART input.

`#include "communication.h"`

Include dependency graph for receive.cpp:

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.

10.37.1 Detailed Description

Implements functions for receiving and processing data, including LoRa and UART input.

Definition in file receive.cpp.

10.37.2 Macro Definition Documentation

10.37.2.1 MAX_PACKET_SIZE

```
#define MAX_PACKET_SIZE 255
```

Definition at line 3 of file receive.cpp.

10.38 receive.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002
00003 #define MAX_PACKET_SIZE 255
00004
00011
00019 void on_receive(int packet_size) {
00020     if (packet_size == 0) return;
00021     uart_print("Received LoRa packet of size " + std::to_string(packet_size), VerbosityLevel::DEBUG);
00022
00023     std::vector<uint8_t> buffer;
00024     buffer.reserve(packet_size);
00025
00026     int bytes_read = 0;
00027
00028     while (LoRa.available() && bytes_read < packet_size) {
00029         if (bytes_read >= MAX_PACKET_SIZE) {
00030             uart_print("Error: Packet exceeds maximum allowed size!", VerbosityLevel::ERROR);
00031             return;
00032         }
00033         buffer.push_back(LoRa.read());
00034         bytes_read++;
00035     }
00036
00037     if (bytes_read < 2) {
00038         uart_print("Error: Packet too small to contain metadata!", VerbosityLevel::ERROR);
00039         return;
00040     }
00041
00042     uart_print("Received " + std::to_string(bytes_read) + " bytes", VerbosityLevel::DEBUG);
00043
00044     uint8_t received_destination = buffer[0];
00045     uint8_t received_local_address = buffer[1];
00046
00047     if (received_destination != lora_address_local) {
00048         uart_print("Error: Destination address mismatch!", VerbosityLevel::ERROR);
00049         return;
00050     }
00051
00052     if (received_local_address != lora_address_remote) {
00053         uart_print("Error: Local address mismatch!", VerbosityLevel::ERROR);
00054         return;
00055     }
00056
00057     // Skip 2 bytes being local and remote address appended by ground station
00058     int start_index = 2;
00059     std::string received(buffer.begin() + start_index, buffer.end());
00060
00061     if (received.empty()) return;
00062
00063     std::stringstream hex_dump;
00064     hex_dump « "Raw bytes: ";
00065     for (int i = 0; i < bytes_read; i++) {
00066         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;
        }
    }
}
```

10.39 lib/comms/send.cpp File Reference

Implements functions for sending data, including LoRa messages and Frames.

`#include "communication.h"`

Include dependency graph for send.cpp:

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)

10.39.1 Detailed Description

Implements functions for sending data, including LoRa messages and Frames.

Definition in file send.cpp.

10.39.2 Function Documentation

10.39.2.1 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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.39.2.2 send_frame_lora()

```
void send_frame_lora (
            const Frame & frame)
```

Definition at line 39 of file send.cpp.

10.39.2.3 send_frame_uart()

```
void send_frame_uart (
            const Frame & frame)
```

Definition at line 47 of file send.cpp.

10.40 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
```

10.41 lib/comms/utils_converters.cpp File Reference

Implements utility functions for converting between different data types.

`#include "communication.h"`

Include dependency graph for utils_converters.cpp:

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.

10.41.1 Detailed Description

Implements utility functions for converting between different data types.

Definition in file utils_converters.cpp.

10.42 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;
}

```

10.43 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"
```

Include dependency graph for event_manager.cpp:

10.43.1 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.

10.44 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() {
    return load_from_storage();
}


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;
}

```

10.45 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"
```

Include dependency graph for event_manager.h:

This graph shows which files directly or indirectly include this file:

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__

10.45.1 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.

10.45.2 Variable Documentation

10.45.2.1 id

```
uint16_t id
```

Unique identifier for the event.

Definition at line 2 of file event_manager.h.

10.45.2.2 timestamp

```
uint32_t timestamp
```

Timestamp of the event in milliseconds since boot.

Definition at line 4 of file event_manager.h.

10.45.2.3 group

```
uint8_t group
```

Event group.

Definition at line 6 of file event_manager.h.

10.45.2.4 event

```
uint8_t event
```

Event code.

Definition at line 8 of file event_manager.h.

10.46 event_manager.h

Go to the documentation of this file.

```
00001
00017
00018 #ifndef EVENT_MANAGER_H
00019 #define EVENT_MANAGER_H
00020
00021 #include "PowerManager.h"
00022 #include <cstdint>
00023 #include <string>
00024 #include "pico/mutex.h"
00025 #include "storage.h"
00026 #include "utils.h"
00027 #include "system_state_manager.h"
00028
00032 #define EVENT_BUFFER_SIZE 100
00033
00037 #define EVENT_FLUSH_THRESHOLD 10
00038
00042 #define EVENT_LOG_FILE "/event_log.csv"
00043
00044
00050 enum class EventGroup : uint8_t {
00052     SYSTEM = 0x00,
00054     POWER = 0x01,
00056     COMMS = 0x02,
00058     GPS = 0x03,
00060     CLOCK = 0x04
00061 };
00062
00063
00069 enum class SystemEvent : uint8_t {
00071     BOOT = 0x01,
00073     SHUTDOWN = 0x02,
00075     WATCHDOG_RESET = 0x03,
00077     CORE1_START = 0x04,
00079     CORE1_STOP = 0x05
00080 };
00081
00087 enum class PowerEvent : uint8_t {
00089     BATTERY_LOW = 0x01,
00091     BATTERY_FULL = 0x02,
00093     POWER_FALLING = 0x03,
00095     BATTERY_NORMAL = 0x04,
00097     SOLAR_ACTIVE = 0x05,
00099     SOLAR_INACTIVE = 0x06,
00101     USB_CONNECTED = 0x07,
00103     USB_DISCONNECTED = 0x08
00104 };
00105
00106
00112 enum class CommsEvent : uint8_t {
00114     RADIO_INIT = 0x01,
00116     RADIO_ERROR = 0x02,
00118     MSG_RECEIVED = 0x03,
00120     MSG_SENT = 0x04,
00122     UART_ERROR = 0x06
00123 };
00124
00130 enum class GPSEvent : uint8_t {
00132     LOCK = 0x01,
00134     LOST = 0x02,
00136     ERROR = 0x03,
00138     POWER_ON = 0x04,
00140     POWER_OFF = 0x05,
00142     DATA_READY = 0x06,
00144     PASS_THROUGH_START = 0x07,
00146     PASS_THROUGH_END = 0x08
00147 };
00148
00154 enum class ClockEvent : uint8_t {
00156     CHANGED = 0x01,
00158     GPS_SYNC = 0x02,
00160     GPS_SYNC_DATA_NOT_READY = 0x03
00161 };
00162
00163
00169 class EventLog {
00170     public:
00172         uint16_t id;
00174         uint32_t timestamp;
00176         uint8_t group;
00178         uint8_t event;
00179     } __attribute__((packed));
```

```
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
```

10.47 lib/location/gps_collector.cpp File Reference

Implementation of the GPS data collector module.

```
#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"
```

Include dependency graph for gps_collector.cpp:

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.

10.47.1 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.

10.48 gps_collector.cpp

Go to the documentation of this file.

```
00001
00014
00015 #include "lib/location/gps_collector.h"
00016 #include "utils.h"
00017 #include "pico/time.h"
00018 #include "lib/location/NMEA/nmea_data.h"
00019 #include "event_manager.h"
00020 #include <vector>
00021 #include <ctime>
00022 #include <cstring>
00023 #include "DS3231.h"
00024 #include <sstream>
00025 #include "system_state_manager.h"
00026
00030 #define MAX_RAW_DATA_LENGTH 256
00031
00040 std::vector<std::string> splitString(const std::string& str, char delimiter) {
00041     std::vector<std::string> tokens;
00042     std::stringstream ss(str);
00043     std::string token;
00044     while (std::getline(ss, token, delimiter)) {
00045         tokens.push_back(token);
00046     }
00047     return tokens;
00048 }
00049
```

```
00059 void collect_gps_data() {
00060
00061     if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
00062         return;
00063     }
00064
00065     static char raw_data_buffer[MAX_RAW_DATA_LENGTH];
00066     static int raw_data_index = 0;
00067
00068     while (uart_is_readable(GPS_UART_PORT)) {
00069         char c = uart_getc(GPS_UART_PORT);
00070
00071         if (c == '\r' || c == '\n') {
00072             // End of message
00073             if (raw_data_index > 0) {
00074                 raw_data_buffer[raw_data_index] = '\0';
00075                 std::string message(raw_data_buffer);
00076                 raw_data_index = 0;
00077
00078                 // Split the message into tokens
00079                 std::vector<std::string> tokens = splitString(message, ',');
00080
00081                 // Update the global vectors based on the sentence type
00082                 if (message.find("$GPRMC") == 0) {
00083                     NMEAData::get_instance().update_rmc_tokens(tokens);
00084                 } else if (message.find("$GPGGA") == 0) {
00085                     NMEAData::get_instance().update_gga_tokens(tokens);
00086                 }
00087             }
00088         } else {
00089             // Append to buffer
00090             if (raw_data_index < MAX_RAW_DATA_LENGTH - 1) {
00091                 raw_data_buffer[raw_data_index++] = c;
00092             } else {
00093                 raw_data_index = 0;
00094             }
00095         }
00096     }
00097 }
```

10.49 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"
```

Include dependency graph for gps_collector.h:

This graph shows which files directly or indirectly include this file:

Functions

- void collect_gps_data ()

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

10.49.1 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.

10.50 gps_collector.h

Go to the documentation of this file.

```
00001
00014
00015  #ifndef GPS_COLLECTOR_H
00016  #define GPS_COLLECTOR_H
00017
00018  #include <string>
00019  #include "hardware/uart.h"
00020  #include "lib/location/NMEA/nmea_data.h" // Include the new header
00021  #include "pin_config.h"
00022
00032  void collect_gps_data();
00033
00034  #endif
```

10.51 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>
```

Include dependency graph for NMEA_data.h:

This graph shows which files directly or indirectly include this file:

Classes

- class NMEAData

  Manages parsed NMEA sentences.

10.51.1 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.

10.52 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
```

10.53 lib/pin_config.h File Reference

`#include <stdint.h>`

Include dependency graph for pin_config.h:

This graph shows which files directly or indirectly include this file:

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 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

10.53.1 Macro Definition Documentation

10.53.1.1 DEBUG_UART_PORT

```
#define DEBUG_UART_PORT uart0
```

Definition at line 8 of file pin_config.h.

10.53.1.2 DEBUG_UART_BAUD_RATE

```
#define DEBUG_UART_BAUD_RATE 115200
```

Definition at line 9 of file pin_config.h.

10.53.1.3 DEBUG_UART_TX_PIN

```
#define DEBUG_UART_TX_PIN 0
```

Definition at line 11 of file pin_config.h.

10.53.1.4 DEBUG_UART_RX_PIN

```
#define DEBUG_UART_RX_PIN 1
```

Definition at line 12 of file pin_config.h.

10.53.1.5 MAIN_I2C_PORT

```
#define MAIN_I2C_PORT i2c1
```

Definition at line 14 of file pin_config.h.

10.53.1.6 MAIN_I2C_SDA_PIN

```
#define MAIN_I2C_SDA_PIN 6
```

Definition at line 15 of file pin_config.h.

10.53.1.7 MAIN_I2C_SCL_PIN

```
#define MAIN_I2C_SCL_PIN 7
```

Definition at line 16 of file pin_config.h.

10.53.1.8 GPS_UART_PORT

```
#define GPS_UART_PORT uart1
```

Definition at line 19 of file pin_config.h.

10.53.1.9 GPS_UART_BAUD_RATE

```
#define GPS_UART_BAUD_RATE 9600
```

Definition at line 20 of file pin_config.h.

10.53.1.10 GPS_UART_TX_PIN

```
#define GPS_UART_TX_PIN 8
```

Definition at line 21 of file pin_config.h.

10.53.1.11 GPS_UART_RX_PIN

```
#define GPS_UART_RX_PIN 9
```

Definition at line 22 of file pin_config.h.

10.53.1.12 GPS_POWER_ENABLE_PIN

```
#define GPS_POWER_ENABLE_PIN 14
```

Definition at line 23 of file pin_config.h.

10.53.1.13 BUFFER_SIZE

```
#define BUFFER_SIZE 85
```

Definition at line 25 of file pin_config.h.

10.53.1.14 SD_SPI_PORT

```
#define SD_SPI_PORT spi1
```

Definition at line 28 of file pin_config.h.

10.53.1.15 SD_MISO_PIN

```
#define SD_MISO_PIN 12
```

Definition at line 29 of file pin_config.h.

10.53.1.16 SD_MOSI_PIN

```
#define SD_MOSI_PIN 11
```

Definition at line 30 of file pin_config.h.

10.53.1.17 SD_SCK_PIN

```
#define SD_SCK_PIN 10
```

Definition at line 31 of file pin_config.h.

10.53.1.18 SD_CS_PIN

```
#define SD_CS_PIN 13
```

Definition at line 32 of file pin_config.h.

10.53.1.19 SD_CARD_DETECT_PIN

```
#define SD_CARD_DETECT_PIN 28
```

Definition at line 33 of file pin_config.h.

10.53.1.20 SX1278_MISO

```
#define SX1278_MISO 16
```

Definition at line 35 of file pin_config.h.

10.53.1.21 SX1278_CS

```
#define SX1278_CS 17
```

Definition at line 36 of file pin_config.h.

10.53.1.22 SX1278_SCK

```
#define SX1278_SCK 18
```

Definition at line 37 of file pin_config.h.

10.53.1.23 SX1278_MOSI

```
#define SX1278_MOSI 19
```

Definition at line 38 of file pin_config.h.

10.53.1.24 SPI_PORT

```
#define SPI_PORT spi0
```

Definition at line 40 of file pin_config.h.

10.53.1.25 READ_BIT

```
#define READ_BIT 0x80
```

Definition at line 41 of file pin_config.h.

10.53.1.26 LORA_DEFAULT_SPI

```
#define LORA_DEFAULT_SPI spi0
```

Definition at line 43 of file pin_config.h.

10.53.1.27 LORA_DEFAULT_SPI_FREQUENCY

```
#define LORA_DEFAULT_SPI_FREQUENCY 8E6
```

Definition at line 44 of file pin_config.h.

10.53.1.28 LORA_DEFAULT_SS_PIN

```
#define LORA_DEFAULT_SS_PIN 17
```

Definition at line 45 of file pin_config.h.

10.53.1.29 LORA_DEFAULT_RESET_PIN

```
#define LORA_DEFAULT_RESET_PIN 22
```

Definition at line 46 of file pin_config.h.

10.53.1.30 LORA_DEFAULT_DIO0_PIN

```
#define LORA_DEFAULT_DIO0_PIN 20
```

Definition at line 47 of file pin_config.h.

10.53.1.31 PA_OUTPUT_RFO_PIN

```
#define PA_OUTPUT_RFO_PIN 11
```

Definition at line 49 of file pin_config.h.

10.53.1.32 PA_OUTPUT_PA_BOOST_PIN

```
#define PA_OUTPUT_PA_BOOST_PIN 12
```

Definition at line 50 of file pin_config.h.

10.53.2 Variable Documentation

10.53.2.1 lora_cs_pin

```
int lora_cs_pin = 17  [inline], [constexpr]
```

Definition at line 52 of file pin_config.h.

10.53.2.2 lora_reset_pin

```
int lora_reset_pin = 22  [inline], [constexpr]
```

Definition at line 53 of file pin_config.h.

10.53.2.3 lora_irq_pin

```
int lora_irq_pin = 28  [inline], [constexpr]
```

Definition at line 54 of file pin_config.h.

10.53.2.4 lora_address_local

```
uint8_t lora_address_local = 37  [inline]
```

Definition at line 56 of file pin_config.h.

10.53.2.5 lora_address_remote

```
uint8_t lora_address_remote = 21  [inline]
```

Definition at line 57 of file pin_config.h.

10.54 pin_config.h

Go to the documentation of this file.

```
00001 // pin_config.h
00002 #ifndef PIN_CONFIG_H
00003 #define PIN_CONFIG_H
00004
00005 #include <stdint.h>
00006
00007 //DEBUG uart
00008 #define DEBUG_UART_PORT uart0
00009 #define DEBUG_UART_BAUD_RATE 115200
00010
00011 #define DEBUG_UART_TX_PIN 0
00012 #define DEBUG_UART_RX_PIN 1
00013
00014 #define MAIN_I2C_PORT i2c1
00015 #define MAIN_I2C_SDA_PIN 6
00016 #define MAIN_I2C_SCL_PIN 7
00017
00018 // GPS configuration
00019 #define GPS_UART_PORT uart1
00020 #define GPS_UART_BAUD_RATE 9600
00021 #define GPS_UART_TX_PIN 8
00022 #define GPS_UART_RX_PIN 9
00023 #define GPS_POWER_ENABLE_PIN 14
00024
00025 #define BUFFER_SIZE 85  // NMEA sentences are usually under 85 chars
00026
00027 // SPI configuration for SD card
00028 #define SD_SPI_PORT spi1
00029 #define SD_MISO_PIN 12
00030 #define SD_MOSI_PIN 11
00031 #define SD_SCK_PIN 10
00032 #define SD_CS_PIN 13
00033 #define SD_CARD_DETECT_PIN 28
00034
00035 #define SX1278_MISO 16
00036 #define SX1278_CS   17
00037 #define SX1278_SCK  18
00038 #define SX1278_MOSI 19
00039
00040 #define SPI_PORT spi0
00041 #define READ_BIT 0x80
00042
00043 #define LORA_DEFAULT_SPI           spi0
00044 #define LORA_DEFAULT_SPI_FREQUENCY 8E6
00045 #define LORA_DEFAULT_SS_PIN        17
00046 #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 // PIN_CONFIG_H
```

10.55 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>
```

Include dependency graph for INA3221.cpp:

10.55.1 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.

10.56 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;
    }
}
```

10.57 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>
```

Include dependency graph for INA3221.h:

This graph shows which files directly or indirectly include this file:

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.

10.57.1 Detailed Description

Header file for the INA3221 triple-channel power monitor driver.

Definition in file INA3221.h.

10.57.2 Enumeration Type Documentation

10.57.2.1 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.

10.57.2.2 ina3221_ch_t

`enum ina3221_ch_t`

Enumerator

INA3221_CH1	
INA3221_CH2	
INA3221_CH3	

Definition at line 23 of file INA3221.h.

10.57.2.3 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.

10.57.2.4 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.

10.57.2.5 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.

10.57.3 Variable Documentation

10.57.3.1 INA3221_CH_NUM

```
const int INA3221_CH_NUM = 3
```

Number of channels in INA3221.

Definition at line 30 of file INA3221.h.

10.57.3.2 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.

10.58 INA3221.h

Go to the documentation of this file.

```
00001 #ifndef BEASTDEVICES_INA3221_H
00002 #define BEASTDEVICES_INA3221_H
00003
00004 #include <stdint.h>
00005 #include <iostream>
00006 #include <hardware/i2c.h>
00007
00012 typedef enum {
00013     INA3221_ADDR40_GND = 0b1000000, // A0 pin -> GND
00014     INA3221_ADDR41_VCC = 0b1000001, // A0 pin -> VCC
00015     INA3221_ADDR42_SDA = 0b1000010, // A0 pin -> SDA
00016     INA3221_ADDR43_SCL = 0b1000011  // A0 pin -> SCL
00017 } ina3221_addr_t;
00018
00023 typedef enum {
00024     INA3221_CH1 = 0,
00025     INA3221_CH2,
00026     INA3221_CH3,
00027 } ina3221_ch_t;
00028
00030 const int INA3221_CH_NUM = 3;
00032 const int SHUNT_VOLTAGE_LSB_UV = 5;
00033
00034
00038 typedef enum {
00039     INA3221_REG_CONF = 0,
00040     INA3221_REG_CH1_SHUNTV,
00041     INA3221_REG_CH1_BUSV,
00042     INA3221_REG_CH2_SHUNTV,
00043     INA3221_REG_CH2_BUSV,
00044     INA3221_REG_CH3_SHUNTV,
00045     INA3221_REG_CH3_BUSV,
00046     INA3221_REG_CH1_CRIT_ALERT_LIM,
00047     INA3221_REG_CH1_WARNING_ALERT_LIM,
00048     INA3221_REG_CH2_CRIT_ALERT_LIM,
00049     INA3221_REG_CH2_WARNING_ALERT_LIM,
00050     INA3221_REG_CH3_CRIT_ALERT_LIM,
00051     INA3221_REG_CH3_WARNING_ALERT_LIM,
00052     INA3221_REG_SHUNTV_SUM,
00053     INA3221_REG_SHUNTV_SUM_LIM,
00054     INA3221_REG_MASK_ENABLE,
00055     INA3221_REG_PWR_VALID_HI_LIM,
00056     INA3221_REG_PWR_VALID_LO_LIM,
00057     INA3221_REG_MANUF_ID = 0xFE,
00058     INA3221_REG_DIE_ID = 0xFF
00059 } ina3221_reg_t;
00060
00065 typedef enum {
00066     INA3221_REG_CONF_CT_140US = 0,
00067     INA3221_REG_CONF_CT_204US,
00068     INA3221_REG_CONF_CT_332US,
00069     INA3221_REG_CONF_CT_588US,
00070     INA3221_REG_CONF_CT_1100US,
00071     INA3221_REG_CONF_CT_2116US,
00072     INA3221_REG_CONF_CT_4156US,
00073     INA3221_REG_CONF_CT_8244US
00074 } ina3221_conv_time_t;
00075
00080 typedef enum {
00081     INA3221_REG_CONF_AVG_1 = 0,
00082     INA3221_REG_CONF_AVG_4,
00083     INA3221_REG_CONF_AVG_16,
00084     INA3221_REG_CONF_AVG_64,
00085     INA3221_REG_CONF_AVG_128,
00086     INA3221_REG_CONF_AVG_256,
00087     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
```

10.59 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"
```

Include dependency graph for PowerManager.cpp:

10.59.1 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.

10.60 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;
}
```

10.61 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"
```

Include dependency graph for PowerManager.h:

This graph shows which files directly or indirectly include this file:

Classes

- class PowerManager

  Manages power-related functions.

10.61.1 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.

10.62 PowerManager.h

Go to the documentation of this file.

```
00001
00014
00015 #ifndef POWER_MANAGER_H
00016 #define POWER_MANAGER_H
00017
00018 #include "INA3221/INA3221.h"
00019 #include <map>
00020 #include <string>
00021 #include <hardware/i2c.h>
00022 #include "pico/stdlib.h"
00023 #include "pico/mutex.h"
00024
00032 class PowerManager {
00033 public:
00038     PowerManager(i2c_inst_t* i2c);
00039
```

```
    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
```

10.63 lib/sensors/BH1750/BH1750.cpp File Reference

```
#include "BH1750.h"
#include "pico/stdlib.h"
#include <stdio.h>
#include <iostream>
```

Include dependency graph for BH1750.cpp:

10.64 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) {} // Initialize i2c_port_

bool BH1750::begin(Mode mode) {
    write8(static_cast<uint8_t>(Mode::POWER_ON));
    write8(static_cast<uint8_t>(Mode::RESET));
    configure(mode);
    configure(BH1750::Mode::POWER_ON);
    uint8_t cmd = 0x10; // Continuously H-Resolution Mode
    if (i2c_write_blocking(i2c_port_, _i2c_addr, &cmd, 1, false) == 1) { // Use i2c_port_
        std::cout « "BH1750 sensor found at 0x" « std::hex « (int)_i2c_addr « std::endl;
        return true;
    }
    return false;
}

void BH1750::configure(Mode mode) {
    uint8_t modeVal = static_cast<uint8_t>(mode);
    switch (mode) {
        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);
            break;
        default:
            printf("Invalid measurement mode\n");
            break;
    }
}

float BH1750::get_light_level() {
    uint8_t buffer[2];
    i2c_read_blocking(i2c_port_, _i2c_addr, buffer, 2, false); // Use i2c_port_
    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); // Use i2c_port_
}
```

10.65 lib/sensors/BH1750/BH1750.h File Reference

`#include "hardware/i2c.h"`
Include dependency graph for BH1750.h:

This graph shows which files directly or indirectly include this file:

Classes

- class BH1750

Macros

- #define _BH1750_DEVICE_ID 0xE1
- #define _BH1750_MTREG_MIN 31
- #define _BH1750_MTREG_MAX 254
- #define _BH1750_DEFAULT_MTREG 69

10.65.1 Macro Definition Documentation

10.65.1.1 _BH1750_DEVICE_ID

```
#define _BH1750_DEVICE_ID 0xE1
```

Definition at line 7 of file BH1750.h.

10.65.1.2 _BH1750_MTREG_MIN

```
#define _BH1750_MTREG_MIN 31
```

Definition at line 8 of file BH1750.h.

10.65.1.3 _BH1750_MTREG_MAX

```
#define _BH1750_MTREG_MAX 254
```

Definition at line 9 of file BH1750.h.

10.65.1.4 _BH1750_DEFAULT_MTREG

```
#define _BH1750_DEFAULT_MTREG 69
```

Definition at line 10 of file BH1750.h.

10.66 BH1750.h

Go to the documentation of this file.

```
00001 #ifndef __BH1750_H__
00002 #define __BH1750_H__
00003
00004 #include "hardware/i2c.h"
00005
00006 // Define constants
00007 #define _BH1750_DEVICE_ID 0xE1  // Correct content of WHO_AM_I register
00008 #define _BH1750_MTREG_MIN 31
00009 #define _BH1750_MTREG_MAX 254
00010 #define _BH1750_DEFAULT_MTREG 69
00011
00012 class BH1750 {
00013 public:
00014     // Scoped enum for measurement modes
00015     enum class Mode : uint8_t {
00016         UNCONFIGURED_POWER_DOWN = 0x00,
00017         POWER_ON = 0x01,
00018         RESET = 0x07,
00019         CONTINUOUS_HIGH_RES_MODE = 0x10,
00020         CONTINUOUS_HIGH_RES_MODE_2 = 0x11,
00021         CONTINUOUS_LOW_RES_MODE = 0x13,
00022         ONE_TIME_HIGH_RES_MODE = 0x20,
00023         ONE_TIME_HIGH_RES_MODE_2 = 0x21,
00024         ONE_TIME_LOW_RES_MODE = 0x23
00025     };
00026
00027     BH1750(i2c_inst_t* i2c, uint8_t addr = 0x23); // Modified constructor
00028     bool begin(Mode mode = Mode::CONTINUOUS_HIGH_RES_MODE);
00029     void configure(Mode mode);
00030     float get_light_level();
00031
00032 private:
00033     void write8(uint8_t data);
00034     uint8_t _i2c_addr;
00035     i2c_inst_t* i2c_port_; // Add i2c_port_ member
00036 };
00037
00038 #endif // __BH1750_H__
```

10.67 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference

```
#include "BH1750_WRAPPER.h"
#include <string>
#include <iostream>
```

Include dependency graph for BH1750_WRAPPER.cpp:

10.68 BH1750_WRAPPER.cpp

Go to the documentation of this file.

```
#include "BH1750_WRAPPER.h"
#include <string>
#include <iostream>

BH1750Wrapper::BH1750Wrapper(i2c_inst_t* i2c) : i2c_port_(i2c), sensor_(i2c, 0x23) { // Initialize i2c_port_ and sensor_
    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;
            }
        }
        // Handle additional configuration keys here
        else {
            std::cerr « "[BH1750Wrapper] Unknown configuration key: " « key « std::endl;
            return false;
        }
    }
    return true;
}
```

10.69 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "BH1750.h"
#include <map>
#include <string>
```

Include dependency graph for BH1750_WRAPPER.h:

This graph shows which files directly or indirectly include this file:

Classes

- class BH1750Wrapper

10.70 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;
    i2c_inst_t* i2c_port_; // Add i2c_port_ member

public:
    BH1750Wrapper(i2c_inst_t* i2c); // Add constructor with i2c_inst_t*
    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
```

10.71 lib/sensors/BME280/BME280.cpp File Reference

```
#include "BME280.h"
#include <iomanip>
```

```
#include <vector>
#include <algorithm>
#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdlib.h"
```

Include dependency graph for BME280.cpp:

10.72 BME280.cpp

Go to the documentation of this file.

```
00001 // BME280.cpp
00002
00003 #include "BME280.h"
00004
00005 #include <iomanip>
00006 #include <vector>
00007 #include <algorithm>
00008 #include "hardware/i2c.h"
00009 #include "pico/binary_info.h"
00010 #include "pico/stdlib.h"
00011
00012 // BME280 (BME280) Class Implementation
00013
00014 BME280::BME280(i2c_inst_t* i2cPort, uint8_t address)
00015     : i2c_port(i2cPort), device_addr(address), calib_params{}, initialized_(false), t_fine(0) {
00016 }
00017
00018 bool BME280::init() {
00019     if (!i2c_port) {
00020         std::cerr « "Invalid I2C port.\n";
00021         return false;
00022     }
00023
00024     // Check device ID to confirm it's a BME280
00025     uint8_t reg = 0xD0; // Chip ID register
00026     uint8_t chip_id = 0;
00027     int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
00028     if (ret != 1) {
00029         std::cerr « "Failed to write to BME280.\n";
00030         return false;
00031     }
00032     ret = i2c_read_blocking(i2c_port, device_addr, &chip_id, 1, false);
00033     if (ret != 1) {
00034         std::cerr « "Failed to read chip ID from BME280.\n";
00035         return false;
00036     }
00037     if (chip_id != 0x60) {
00038         std::cerr « "Device is not a BME280.\n";
00039         return false;
00040     }
00041
00042     // Configure sensor
00043     if (!configure_sensor()) {
00044         std::cerr « "Failed to configure BME280 sensor.\n";
00045         return false;
00046     }
00047
00048     // Retrieve calibration parameters
00049     if (!get_calibration_parameters()) {
00050         std::cerr « "Failed to retrieve calibration parameters.\n";
```

```
        return false;
    }

    initialized_ = true;
    std::cout « "BME280 sensor initialized_ successfully.\n";
    return true;
}

void BME280::reset() {
    uint8_t buf[2] = { REG_RESET, 0xB6 };
    int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
    if (ret != 2) {
        std::cerr « "Failed to reset BME280 sensor.\n";
    }
    sleep_ms(10); // Wait for reset to complete
}

bool BME280::read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity) {
    if (!initialized_) {
        std::cerr « "BME280 not initialized_.\n";
        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
    int ret = i2c_write_blocking(i2c_port, device_addr, &start_reg, 1, true);
    if (ret != 1) {
        std::cerr « "Failed to write starting register address to BME280.\n";
        return false;
    }

    // Read data
    ret = i2c_read_blocking(i2c_port, device_addr, buf, 8, false);
    if (ret != 8) {
        std::cerr « "Failed to read data from BME280.\n";
        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[26];
    uint8_t reg = REG_DIG_T1_LSB;
    int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
    if (ret != 1) {
        std::cerr « "Failed to write to BME280.\n";
        return false;
    }
    ret = i2c_read_blocking(i2c_port, device_addr, calib_data, 26, false);
    if (ret != 26) {
        std::cerr « "Failed to read calibration data from BME280.\n";
        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)
    reg = 0xE1;
    ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
    if (ret != 1) {
        std::cerr « "Failed to write to BME280 for humidity calibration.\n";
        return false;
    }

    uint8_t hum_calib_data[7];
    ret = i2c_read_blocking(i2c_port, device_addr, hum_calib_data, 7, false);
    if (ret != 7) {
        std::cerr « "Failed to read humidity calibration data from BME280.\n";
        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() {
    uint8_t buf[2];

    // Set humidity oversampling (must be set before ctrl_meas)
    buf[0] = REG_CTRL_HUM;
    buf[1] = 0x05; // Humidity oversampling x16
    int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
    if (ret != 2) {
        std::cerr « "Failed to write CTRL_HUM to BME280.\n";
        return false;
    }

    // Write config register
    buf[0] = REG_CONFIG;
    buf[1] = 0x00; // Default settings
    ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
    if (ret != 2) {
        std::cerr « "Failed to write CONFIG to BME280.\n";
        return false;
```

```
    }

    // Write ctrl_meas register
    buf[0] = REG_CTRL_MEAS;
    buf[1] = 0xB7; // Temp and pressure oversampling x16, normal mode
    ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
    if (ret !=2) {
        std::cerr « "Failed to write CTRL_MEAS to BME280.\n";
        return false;
    }

    return true;
}
```

10.73 lib/sensors/BME280/BME280.h File Reference

```
#include <cstdint>
#include <iostream>
#include "hardware/i2c.h"
```

Include dependency graph for BME280.h:

This graph shows which files directly or indirectly include this file:

Classes

- struct BME280CalibParam
- class BME280

10.74 BME280.h

Go to the documentation of this file.

```
// BME280.h

#ifndef BME280_H
#define BME280_H

#include <cstdint>
#include <iostream>
#include "hardware/i2c.h"

// Calibration parameters structure
struct BME280CalibParam {
    // Temperature parameters
    uint16_t dig_t1;
    int16_t  dig_t2;
    int16_t  dig_t3;

    // Pressure parameters
    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;

    // Humidity parameters
    uint8_t  dig_h1;
    int16_t  dig_h2;
    uint8_t  dig_h3;
    int16_t  dig_h4;
    int16_t  dig_h5;
    int8_t   dig_h6;
};

// BME280 Class Definition
class BME280 {
public:
    // I2C Address Options
    static constexpr uint8_t ADDR_SDO_LOW = 0x76;
    static constexpr uint8_t ADDR_SDO_HIGH = 0x77;

    // Constructor
    BME280(i2c_inst_t* i2cPort, uint8_t address = ADDR_SDO_LOW);

    // Initialize the sensor
    bool init();

    // Reset the sensor
    void reset();

    // Read all raw data: temperature, pressure, and humidity
    bool read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity);

    // Convert raw data to actual values
    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:
    // Configure the sensor
    bool configure_sensor();

    // Retrieve calibration parameters from the sensor
    bool get_calibration_parameters();

    // I2C port and device address
    i2c_inst_t* i2c_port;
    uint8_t device_addr;

```

```
    // Calibration parameters
    BME280CalibParam calib_params;

    // Initialization status
    bool initialized_;

    // Fine temperature parameter needed for compensation
    mutable int32_t t_fine;

    // Register Definitions
    static constexpr uint8_t REG_CONFIG            = 0xF5;
    static constexpr uint8_t REG_CTRL_MEAS         = 0xF4;
    static constexpr uint8_t REG_CTRL_HUM          = 0xF2;
    static constexpr uint8_t REG_RESET             = 0xE0;

    static constexpr uint8_t REG_PRESSURE_MSB      = 0xF7;
    static constexpr uint8_t REG_TEMPERATURE_MSB   = 0xFA;
    static constexpr uint8_t REG_HUMIDITY_MSB      = 0xFD;

    // Calibration Registers
    static constexpr uint8_t REG_DIG_T1_LSB        = 0x88;
    static constexpr uint8_t REG_DIG_T1_MSB        = 0x89;
    static constexpr uint8_t REG_DIG_T2_LSB        = 0x8A;
    static constexpr uint8_t REG_DIG_T2_MSB        = 0x8B;
    static constexpr uint8_t REG_DIG_T3_LSB        = 0x8C;
    static constexpr uint8_t REG_DIG_T3_MSB        = 0x8D;

    static constexpr uint8_t REG_DIG_P1_LSB        = 0x8E;
    static constexpr uint8_t REG_DIG_P1_MSB        = 0x8F;
    static constexpr uint8_t REG_DIG_P2_LSB        = 0x90;
    static constexpr uint8_t REG_DIG_P2_MSB        = 0x91;
    static constexpr uint8_t REG_DIG_P3_LSB        = 0x92;
    static constexpr uint8_t REG_DIG_P3_MSB        = 0x93;
    static constexpr uint8_t REG_DIG_P4_LSB        = 0x94;
    static constexpr uint8_t REG_DIG_P4_MSB        = 0x95;
    static constexpr uint8_t REG_DIG_P5_LSB        = 0x96;
    static constexpr uint8_t REG_DIG_P5_MSB        = 0x97;
    static constexpr uint8_t REG_DIG_P6_LSB        = 0x98;
    static constexpr uint8_t REG_DIG_P6_MSB        = 0x99;
    static constexpr uint8_t REG_DIG_P7_LSB        = 0x9A;
    static constexpr uint8_t REG_DIG_P7_MSB        = 0x9B;
    static constexpr uint8_t REG_DIG_P8_LSB        = 0x9C;
    static constexpr uint8_t REG_DIG_P8_MSB        = 0x9D;
    static constexpr uint8_t REG_DIG_P9_LSB        = 0x9E;
    static constexpr uint8_t REG_DIG_P9_MSB        = 0x9F;

    // Humidity Calibration Registers
    static constexpr uint8_t REG_DIG_H1            = 0xA1;
    static constexpr uint8_t REG_DIG_H2            = 0xE1;
    static constexpr uint8_t REG_DIG_H3            = 0xE3;
    static constexpr uint8_t REG_DIG_H4            = 0xE4;
    static constexpr uint8_t REG_DIG_H5            = 0xE5;
    static constexpr uint8_t REG_DIG_H6            = 0xE7;

    // Number of calibration parameters to read
    static constexpr size_t NUM_CALIB_PARAMS = 24;
};

#endif // BME280_H
```

10.75 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference

`#include "BME280_WRAPPER.h"`

Include dependency graph for BME280_WRAPPER.cpp:

10.76 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(const std::map<std::string, std::string>& config) {
    return true;
}
```

10.77 lib/sensors/BME280/BME280_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "BME280.h"
```

Include dependency graph for BME280_WRAPPER.h:

This graph shows which files directly or indirectly include this file:

Classes

- class BME280Wrapper

10.78 BME280_WRAPPER.h

Go to the documentation of this file.

```
00001 // BME280_WRAPPER.h
00002 #ifndef BME280_WRAPPER_H
00003 #define BME280_WRAPPER_H
00004
00005 #include "ISensor.h"
00006 #include "BME280.h"
00007
00008 class BME280Wrapper : public ISensor {
00009 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
```

10.79 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"
```

Include dependency graph for ISensor.cpp:

10.79.1 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.

10.80 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;
}
```

10.81 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"
```

Include dependency graph for ISensor.h:

This graph shows which files directly or indirectly include this file:

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.

10.81.1 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.

10.82 ISensor.h

Go to the documentation of this file.

```
00001
00016
00017 #ifndef ISENSOR_H
00018 #define ISENSOR_H
00019
00020 #include <map>
00021 #include <string>
00022 #include <vector>
00023 #include <utility>
00024 #include "hardware/i2c.h"
00025
00031 enum class SensorType : uint8_t {
00033     NONE = 0x00,
00035     LIGHT = 0x01,
00037     ENVIRONMENT = 0x02,
00038 };
00039
00045 enum class SensorDataTypeIdentifier : uint8_t {
00047     NONE = 0x00,
00049     LIGHT_LEVEL = 0x01,
00051     TEMPERATURE = 0x02,
00053     HUMIDITY = 0x03,
00055     PRESSURE = 0x04,
00056 };
00057
00063 class ISensor {
00064 public:
00069     virtual ~ISensor() = default;
00070
00075     virtual bool init() = 0;
00076
00082     virtual float read_data(SensorDataTypeIdentifier type) = 0;
00083
00088     virtual bool is_initialized() const = 0;
00089
00094     virtual SensorType get_type() const = 0;
00095
00101     virtual bool configure(const std::map<std::string, std::string>& config) = 0;
00102
00107     virtual uint8_t get_address() const = 0;
00108 };
00109
00116 class SensorWrapper {
00117 public:
00122     static SensorWrapper& get_instance() {
00123         static SensorWrapper instance;
00124         return instance;
00125     }
00126
00133     bool sensor_init(SensorType type, i2c_inst_t* i2c = nullptr);
00134
00141     bool sensor_configure(SensorType type, const std::map<std::string, std::string>& config);
00142
00149     float sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType);
00150
00156     ISensor* get_sensor(SensorType type);
00157
00163     std::vector<std::pair<SensorType, uint8_t>> scan_connected_sensors(i2c_inst_t* i2c);
00164
00169     std::vector<std::pair<SensorType, uint8_t>> get_available_sensors();
00170
00171 private:
00173     std::map<SensorType, ISensor*> sensors;
```

```
    SensorWrapper() = default;
};

#endif
```

10.83 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"
```

Include dependency graph for storage.cpp:

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.

10.83.1 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.

10.84 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;
}
```

10.85 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"
```

Include dependency graph for storage.h:

This graph shows which files directly or indirectly include this file:

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.

10.85.1 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.

10.86 storage.h

Go to the documentation of this file.

```
00001
00013
00014 #ifndef STORAGE_H
00015 #define STORAGE_H
00016
00017 #include <stdio.h>
00018 #include <string.h>
00019 #include <hardware/clocks.h>
00020 #include <hardware/flash.h>
00021 #include "blockdevice/flash.h"
00022 #include "blockdevice/sd.h"
00023 #include "filesystem/littlefs.h"
00024 #include "filesystem/vfs.h"
00025 #include "pin_config.h"
00026 #include "lfs.h"
00027 #include "filesystem/fat.h"
00028
00036 bool fs_init(void);
00037
00043 bool fs_stop(void);
00044
00045 #endif
```

10.87 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"
```

Include dependency graph for system_state_manager.h:

This graph shows which files directly or indirectly include this file:

Classes

- class SystemStateManager

  Manages the system state of the Kubisat firmware.

10.87.1 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.

10.88 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;
        recursive_mutex_t mutex_;

        SystemStateManager() :
            pending_bootloader_reset(false),
            gps_collection_paused(false),
            sd_card_mounted(false),
            uart_verbosity(VerbosityLevel::DEBUG)
        {
            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_);
        }
    };

#endif // SYSTEM_STATE_MANAGER_H
```

10.89 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"
```

Include dependency graph for telemetry_manager.cpp:

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

- bool telemetry_init ()

  Initialize the telemetry system.
- 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 collect_telemetry ()

  Collect telemetry data from sensors and power subsystems.
- 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.

Variables

- static uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS

  Current sampling interval in milliseconds.
- static 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.
- size_t sensor_data_buffer_count = 0
- size_t sensor_data_buffer_write_index = 0
- mutex_t telemetry_mutex

  Mutex for thread-safe access to the telemetry buffer.

10.89.1 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.

10.90 telemetry_manager.cpp

Go to the documentation of this file.

```
00001
00009
00010  #include "telemetry_manager.h"
00011 #include "utils.h"
00012 #include "storage.h"
00013 #include "PowerManager.h"
00014 #include "ISensor.h"
00015 #include "DS3231.h"
00016 #include <deque>
00017 #include <mutex>
00018 #include <iomanip>
00019 #include <sstream>
00020 #include <cstdio>
00021 #include "communication.h"
00022 #include "system_state_manager.h"
00023
00027 #define TELEMETRY_CSV_PATH "/telemetry.csv"
00028
00032 #define SENSOR_DATA_CSV_PATH "/sensors.csv"
00033
00037 #define DEFAULT_SAMPLE_INTERVAL_MS 1000
00038
00042 #define DEFAULT_FLUSH_THRESHOLD 10
00043
00047 static uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS;
00048
00052 static uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD;
00053
00057 TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE];
00058 size_t telemetry_buffer_count = 0;
00059 size_t telemetry_buffer_write_index = 0;
00060
00064 SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE];
00065 size_t sensor_data_buffer_count = 0;
00066 size_t sensor_data_buffer_write_index = 0;
00067
00071 mutex_t telemetry_mutex;
00072
00080 bool telemetry_init() {
00081     mutex_init(&telemetry_mutex);
00082
00083     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00084         bool success = true;
00085
00086         FILE* file = fopen(TELEMETRY_CSV_PATH, "w");
00087         if (!file) {
00088             file = fopen(TELEMETRY_CSV_PATH, "w");
00089             if (file) {
00090                 fprintf(file, "timestamp,build,battery_v,system_v,usb_ma,solar_ma,discharge_ma,"
00091
      "gps_time,latitude,lat_dir,longitude,lon_dir,speed_knots,course_deg,date,"
00092                                 "fix_quality,satellites,altitude_m\n");
00093                 fclose(file);
00094                 uart_print("Created new telemetry log", VerbosityLevel::INFO);
00095             } else {
00096                 uart_print("Failed to create telemetry log", VerbosityLevel::ERROR);
00097                 success = false;
00098             }
00099         } else {
00100             fclose(file);
00101         }
00102
00103         file = fopen(SENSOR_DATA_CSV_PATH, "w");
00104         if (!file) {
00105             file = fopen(SENSOR_DATA_CSV_PATH, "w");
00106             if (file) {
00107                 fprintf(file, "timestamp,temperature,pressure,humidity,light\n");
00108                 fclose(file);
00109                 uart_print("Created new sensor data log", VerbosityLevel::INFO);
00110             } else {
00111                 uart_print("Failed to create sensor data log", VerbosityLevel::ERROR);
00112                 success = false;
00113             }
00114         } else {
00115             fclose(file);
00116         }
00117
00118         return success;
00119     }
00120
00121     uart_print("Telemetry system initialized (storage not available)", VerbosityLevel::WARNING);
00122     return false;
```

```
}


void 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 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 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];
        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 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 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 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* file = fopen(TELEMETRY_CSV_PATH, "a");
    if (!file) {
        uart_print("Failed to open telemetry log for writing", VerbosityLevel::ERROR);
        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(file, "%s\n", telemetry_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(file);

    mutex_exit(&telemetry_mutex);
    return true;
}


bool flush_sensor_data() {
    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* file = fopen(SENSOR_DATA_CSV_PATH, "a");
    if (!file) {
        uart_print("Failed to open sensor data log for writing", VerbosityLevel::ERROR);
        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(file, "%s\n", sensor_data_buffer[read_index].to_csv().c_str());
        read_index = (read_index + 1) % TELEMETRY_BUFFER_SIZE;
    }

    // We don't need to clear the buffer here since it's cleared in flush_telemetry()
    // and both buffers use the same indices

    fclose(file);

    mutex_exit(&telemetry_mutex);
    return true;
}


bool 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 is_telemetry_flush_time(uint32_t& collection_counter) {
    if (collection_counter >= flush_threshold) {
        collection_counter = 0;
        return true;
    }
    return false;
}

std::string get_last_telemetry_record_csv() {
    mutex_enter_blocking(&telemetry_mutex);

    if (telemetry_buffer_count == 0) {
        mutex_exit(&telemetry_mutex);
        return "";
    }

    size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) %
TELEMETRY_BUFFER_SIZE;

    TelemetryRecord last_record = telemetry_buffer[last_record_index];

    mutex_exit(&telemetry_mutex);

    return last_record.to_csv();
}

std::string get_last_sensor_record_csv() {
    mutex_enter_blocking(&telemetry_mutex);

```

```
    if (telemetry_buffer_count == 0) {
        mutex_exit(&telemetry_mutex);
        return "";
    }

    size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) %
TELEMETRY_BUFFER_SIZE;

    SensorDataRecord last_record = sensor_data_buffer[last_record_index];

    mutex_exit(&telemetry_mutex);

    return last_record.to_csv();
}
```

10.91 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>
```

Include dependency graph for telemetry_manager.h:

This graph shows which files directly or indirectly include this file:

Classes

- struct TelemetryRecord

  Structure representing a single telemetry data point.
- struct SensorDataRecord

  Structure representing a single sensor data point.

Functions

- bool telemetry_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.

Variables

- constexpr int TELEMETRY_BUFFER_SIZE = 20

  Circular buffer for telemetry records.
- TelemetryRecord telemetry_buffer [TELEMETRY_BUFFER_SIZE]

  Circular buffer for telemetry records.
- size_t telemetry_buffer_count
- size_t telemetry_buffer_write_index
- SensorDataRecord sensor_data_buffer [TELEMETRY_BUFFER_SIZE]

  Circular buffer for sensor data records.
- size_t sensor_data_buffer_count
- size_t sensor_data_buffer_write_index
- mutex_t telemetry_mutex

  Mutex for thread-safe access to the telemetry buffer.

10.91.1 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.

10.92 telemetry_manager.h

Go to the documentation of this file.

```
00001
00015
00016
00017 #ifndef TELEMETRY_MANAGER_H
00018 #define TELEMETRY_MANAGER_H
00019
00020 #include <cstdint>
00021 #include <string>
00022 #include "pico/stdlib.h"
00023 #include "lib/location/NMEA/nmea_data.h"
00024 #include "utils.h"
00025 #include "storage.h"
00026 #include "PowerManager.h"
00027 #include "ISensor.h"
00028 #include "DS3231.h"
00029 #include <deque>
00030 #include <mutex>
00031 #include <iomanip>
00032 #include <sstream>
00033 #include <cstdio>
00034 #include "communication.h"
00035 #include <functional>
00036
00043 struct TelemetryRecord {
00044     uint32_t timestamp;
00045
00046     std::string build_version;
00047
00048     // Power data
00049     float battery_voltage;
00050     float system_voltage;
00051     float charge_current_usb;
00052     float charge_current_solar;
00053     float discharge_current;
00054
00055     // GPS data - key RMC fields
00056     std::string time;
00057     std::string latitude;
00058     std::string lat_dir;
00059     std::string longitude;
00060     std::string lon_dir;
00061     std::string speed;
00062     std::string course;
00063     std::string date;
00064
00065     // GPS data - key GGA fields
00066     std::string fix_quality;
00067     std::string satellites;
00068     std::string altitude;
00069
00070
00076     std::string to_csv() const {
00077         std::stringstream ss;
00078         ss « timestamp « ","
00079            « build_version « ","
00080            « std::fixed « std::setprecision(3)
00081            « battery_voltage « ","
00082            « system_voltage « ","
00083            « charge_current_usb « ","
00084            « 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();
    }
};

constexpr int TELEMETRY_BUFFER_SIZE = 20;
extern const int TELEMETRY_BUFFER_SIZE;
extern TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE];
extern size_t telemetry_buffer_count;
extern size_t telemetry_buffer_write_index;

extern SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE];
extern size_t sensor_data_buffer_count;
extern size_t sensor_data_buffer_write_index;

extern mutex_t telemetry_mutex;


bool telemetry_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();

#endif // TELEMETRY_MANAGER_H
 // End of TelemetryManager group
```

10.93 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"
```

Include dependency graph for utils.cpp:

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.

10.93.1 Detailed Description

Implementation of utility functions for the Kubisat firmware.

Definition in file utils.cpp.

10.93.2 Function Documentation

10.93.2.1 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.

Here is the caller graph for this function:

10.93.2.2 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.

Here is the caller graph for this function:

10.93.2.3 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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.93.3 Variable Documentation

10.93.3.1 uart_mutex

```
mutex_t uart_mutex  [static]
```

Mutex for UART access protection.

Definition at line 17 of file utils.cpp.

10.94 utils.cpp

Go to the documentation of this file.

```
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
#include <vector>
#include <queue>
#include <string>
#include <array>
#include "system_state_manager.h"


static mutex_t uart_mutex;


std::string get_level_color(VerbosityLevel level) {
    switch (level) {
        case VerbosityLevel::ERROR:   return ANSI_RED;
        case VerbosityLevel::WARNING: return ANSI_YELLOW;
        case VerbosityLevel::INFO:    return ANSI_GREEN;
        case VerbosityLevel::DEBUG:   return ANSI_BLUE;
        default:                      return "";
    }
}


std::string get_level_prefix(VerbosityLevel level) {
    switch (level) {
        case VerbosityLevel::ERROR:   return "ERROR: ";
        case VerbosityLevel::WARNING: return "WARNING: ";
        case VerbosityLevel::INFO:    return "INFO: ";
        case VerbosityLevel::DEBUG:   return "DEBUG: ";
        default:                      return "";
    }
}

void uart_print(const std::string& msg, VerbosityLevel level, uart_inst_t* uart) {
    if (static_cast<int>(level) > static_cast<int>(SystemStateManager::get_instance().get_uart_verbosity())) {
        return;
    }

    static bool mutex_inited = false;
    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);
}
```

10.95 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>
```

Include dependency graph for utils.h:

This graph shows which files directly or indirectly include this file:

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.

10.95.1 Detailed Description

Utility functions and definitions for the Kubisat firmware.

Contains UART logging, color definitions, and CRC calculations

Definition in file utils.h.

10.95.2 Macro Definition Documentation

10.95.2.1 ANSI_RED

```
#define ANSI_RED "\033[31m"
```

ANSI escape codes for terminal color output.

Definition at line 20 of file utils.h.

10.95.2.2 ANSI_GREEN

```
#define ANSI_GREEN "\033[32m"
```

Definition at line 21 of file utils.h.

10.95.2.3 ANSI_YELLOW

```
#define ANSI_YELLOW "\033[33m"
```

Definition at line 22 of file utils.h.

10.95.2.4 ANSI_BLUE

```
#define ANSI_BLUE "\033[34m"
```

Definition at line 23 of file utils.h.

10.95.2.5 ANSI_RESET

```
#define ANSI_RESET "\033[0m"
```

Definition at line 24 of file utils.h.

10.95.3 Enumeration Type Documentation

10.95.3.1 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.

10.95.4 Function Documentation

10.95.4.1 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
logToFile	Whether to store the message in log storage
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.

Here is the call graph for this function:

Here is the caller graph for this function:

10.96 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
};

```

```
00038
00046 void uart_print(const std::string& msg,
00047                 VerbosityLevel level = VerbosityLevel::INFO,
00048                 uart_inst_t* uart = DEBUG_UART_PORT);
00049
00050
00051 #endif
```

10.97 main.cpp File Reference

`#include "includes.h"`

Include dependency graph for main.cpp:

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

10.97.1 Macro Definition Documentation

10.97.1.1 LOG_FILENAME

```
#define LOG_FILENAME "/log.txt"
```

Definition at line 3 of file main.cpp.

10.97.2 Function Documentation

10.97.2.1 core1_entry()

```
void core1_entry ()
```

Definition at line 8 of file main.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.97.2.2 init_pico_hw()

```
bool init_pico_hw ()
```

Definition at line 53 of file main.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.97.2.3 init_modules()

```
bool init_modules ()
```

Definition at line 90 of file main.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.97.2.4 main()

```
int main (
            void )
```

Definition at line 146 of file main.cpp.

Here is the call graph for this function:

10.97.3 Variable Documentation

10.97.3.1 buffer

```
char buffer[BUFFER_SIZE] = {0}
```

Definition at line 5 of file main.cpp.

10.97.3.2 buffer_index

```
int buffer_index = 0
```

Definition at line 6 of file main.cpp.

10.98 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;

    telemetry_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 (is_telemetry_collection_time(currentTime, last_telemetry_time)) {
            collect_telemetry();
            telemetry_collection_counter++;

            if (is_telemetry_flush_time(telemetry_collection_counter)) {
                flush_telemetry();
                flush_sensor_data();
            }
        }

        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);

    SystemStateManager::get_instance();

```

```
    EventEmitter::emit(EventGroup::GPS, GPSEvent::POWER_ON);

    system("color");

    return true;
}

bool init_modules(){
    bool radio_init_status = false;
    radio_init_status = initialize_radio();

    bool sd_init_status = fs_init();
    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, MAIN_I2C_PORT);
    // bool env_sensor_init = sensor_wrapper.sensor_init(SensorType::ENVIRONMENT, MAIN_I2C_PORT);
    // bool mag_sensor_init = sensor_wrapper.sensor_init(SensorType::MAGNETOMETER, MAIN_I2C_PORT);

    // if (!light_sensor_init || !env_sensor_init || !mag_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;
}
```

10.99 refactor.md File Reference

10.100 test/comms/commands/test_clock_commands.cpp File Reference

10.101 test_clock_commands.cpp

Go to the documentation of this file.

```
00001
```

10.102 test/comms/commands/test_diagnostic_commands.cpp File Reference

```
#include "unity.h"
#include "commands.h"
#include "protocol.h"
#include "build_number.h"
```

Include dependency graph for test_diagnostic_commands.cpp:

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)

10.102.1 Function Documentation

10.102.1.1 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
            void )
```

Definition at line 6 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.102.1.2 test_handle_get_build_version()

```
void test_handle_get_build_version (
            void )
```

Definition at line 15 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.102.1.3 test_handle_verbosity()

```
void test_handle_verbosity (
            void )
```

Definition at line 25 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.102.1.4 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
            void )
```

Definition at line 40 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.103 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);
```

```
    TEST_ASSERT_EQUAL(1, response.size());
    TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
    TEST_ASSERT_EQUAL(1, response[0].group);
    TEST_ASSERT_EQUAL(8, response[0].command);
    TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());

    // Test GET operation
    response = handle_verbosity("", OperationType::GET);
    TEST_ASSERT_EQUAL(1, response.size());
    TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
}

void test_handle_enter_bootloader_mode(void) {
    std::vector<Frame> response = handle_enter_bootloader_mode("", OperationType::SET);

    TEST_ASSERT_EQUAL(1, response.size());
    TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
    TEST_ASSERT_EQUAL(1, response[0].group);
    TEST_ASSERT_EQUAL(9, response[0].command);
}
```

10.104 test/comms/commands/test_event_commands.cpp File Reference

10.105 test_event_commands.cpp

Go to the documentation of this file.

```
00001
```

10.106 test/comms/commands/test_gps_commands.cpp File Reference

10.107 test_gps_commands.cpp

Go to the documentation of this file.

```
00001
```

10.108 test/comms/commands/test_power_commands.cpp File Reference

10.109 test_power_commands.cpp

Go to the documentation of this file.

```
00001
```

10.110 test/comms/commands/test_sensor_commands.cpp File Reference

10.111 test_sensor_commands.cpp

Go to the documentation of this file.

```
00001
```

10.112 test/comms/commands/test_storage_commands.cpp File Reference

10.113 test_storage_commands.cpp

Go to the documentation of this file.

```
00001
```

10.114 test/comms/commands/test_telemetry_commands.cpp File Reference

10.115 test_telemetry_commands.cpp

Go to the documentation of this file.

```
00001
```

10.116 test/comms/test_comand_handlers.cpp File Reference

```
#include "unity.h"
#include "protocol.h"
#include "communication.h"
#include "commands.h"
```

Include dependency graph for test_comand_handlers.cpp:

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

10.116.1 Function Documentation

10.116.1.1 send_frame_uart()

```
void send_frame_uart (
            const Frame & frame)
```

Definition at line 11 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.116.1.2 send_frame_lora()

```
void send_frame_lora (
            const Frame & frame)
```

Definition at line 16 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.116.1.3 setUp()

```
void setUp (
            void )
```

Definition at line 21 of file test_comand_handlers.cpp.

10.116.1.4 tearDown()

```
void tearDown (
            void )
```

Definition at line 26 of file test_comand_handlers.cpp.

10.116.1.5 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
            void )
```

Definition at line 29 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.116.1.6 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
            void )
```

Definition at line 39 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.116.1.7 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
            void )
```

Definition at line 54 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.116.2 Variable Documentation

10.116.2.1 uart_send_called

```
bool uart_send_called = false  [static]
```

Definition at line 7 of file test_comand_handlers.cpp.

10.116.2.2 lora_send_called

```
bool lora_send_called = false  [static]
```

Definition at line 8 of file test_comand_handlers.cpp.

10.116.2.3 last_frame_sent

```
Frame last_frame_sent  [static]
```

Definition at line 9 of file test_comand_handlers.cpp.

10.117 test_comand_handlers.cpp

Go to the documentation of this file.

```
// test/comms/test_command_handlers.cpp
#include "unity.h"
#include "protocol.h"
#include "communication.h"
#include "commands.h"

static bool uart_send_called = false;
static bool lora_send_called = false;
static Frame last_frame_sent;

void send_frame_uart(const Frame& frame) {
    uart_send_called = true;
    last_frame_sent = frame;
}

void send_frame_lora(const Frame& frame) {
    lora_send_called = true;
    last_frame_sent = frame;
}

void setUp(void) {
    uart_send_called = false;
    lora_send_called = false;
}

void tearDown(void) {
}

void test_command_handler_get_operation(void) {
    std::vector<Frame> response = handle_get_build_version("", OperationType::GET);

    TEST_ASSERT_EQUAL(1, response.size());
    TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
    TEST_ASSERT_EQUAL(1, response[0].group);
    TEST_ASSERT_EQUAL(1, response[0].command);
    TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
}

void test_command_handler_set_operation(void) {
    VerbosityLevel old_level = get_verbosity_level();
    std::vector<Frame> response = handle_verbosity("2", OperationType::SET);

    TEST_ASSERT_EQUAL(1, response.size());
    TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
    TEST_ASSERT_EQUAL(1, response[0].group);
    TEST_ASSERT_EQUAL(8, response[0].command);
    TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());

    TEST_ASSERT_EQUAL(VerbosityLevel::WARNING, get_verbosity_level());

    set_verbosity_level(old_level);
}

void test_command_handler_invalid_operation(void) {
    std::vector<Frame> response = handle_get_build_version("", OperationType::SET);

    TEST_ASSERT_EQUAL(1, response.size());
    TEST_ASSERT_EQUAL(OperationType::ERR, response[0].operationType);
    TEST_ASSERT_EQUAL(1, response[0].group);
    TEST_ASSERT_EQUAL(1, response[0].command);
}
```

10.118 test/comms/test_converters.cpp File Reference

`#include "test_frame_common.h"`

Include dependency graph for test_converters.cpp:

Functions

- void test_operation_type_conversion ()
- void test_value_unit_type_conversion ()
- void test_exception_type_conversion ()
- void test_hex_string_conversion ()

10.118.1 Function Documentation

10.118.1.1 test_operation_type_conversion()

```
void test_operation_type_conversion (
            void )
```

Definition at line 4 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.118.1.2 test_value_unit_type_conversion()

```
void test_value_unit_type_conversion (
            void )
```

Definition at line 13 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.118.1.3 test_exception_type_conversion()

```
void test_exception_type_conversion (
            void )
```

Definition at line 20 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.118.1.4 test_hex_string_conversion()

```
void test_hex_string_conversion (
            void )
```

Definition at line 27 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.119 test_converters.cpp

Go to the documentation of this file.

```
00001 // test_frame_converters.cpp
00002 #include "test_frame_common.h"
00003
00004 void test_operation_type_conversion() {
00005     OperationType type = OperationType::GET;
00006     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]);
}
```

10.120 test/comms/test_frame_build.cpp File Reference

`#include "test_frame_common.h"`

Include dependency graph for test_frame_build.cpp:

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 ()

10.120.1 Function Documentation

10.120.1.1 test_frame_build_val()

```
void test_frame_build_val (
            void )
```

Definition at line 4 of file test_frame_build.cpp.

Here is the call graph for this function:

10.120.1.2 test_frame_build_err()

```
void test_frame_build_err (
            void )
```

Definition at line 15 of file test_frame_build.cpp.

Here is the call graph for this function:

10.120.1.3 test_frame_build_get()

```
void test_frame_build_get (
            void )
```

Definition at line 24 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.120.1.4 test_frame_build_set()

```
void test_frame_build_set (
            void )
```

Definition at line 35 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.120.1.5 test_frame_build_res()

```
void test_frame_build_res (
            void )
```

Definition at line 46 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.120.1.6 test_frame_build_seq()

```
void test_frame_build_seq (
            void )
```

Definition at line 57 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.121 test_frame_build.cpp

Go to the documentation of this file.

```cpp
// 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"
}
```

10.122 test/comms/test_frame_coding.cpp File Reference

```
#include "test_frame_common.h"
```

Include dependency graph for test_frame_coding.cpp:

Functions

- void test_frame_encode_basic ()
- void test_frame_decode_basic ()
- void test_frame_decode_invalid_header ()

10.122.1 Function Documentation

10.122.1.1 test_frame_encode_basic()

```
void test_frame_encode_basic (
            void )
```

Definition at line 4 of file test_frame_coding.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.122.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
            void )
```

Definition at line 14 of file test_frame_coding.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.122.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
            void )
```

Definition at line 26 of file test_frame_coding.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.123 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     Frame frame = create_test_frame();
00006     std::string encoded = frame_encode(frame);
00007
00008     TEST_ASSERT_NOT_EQUAL(0, encoded.length());
00009     TEST_ASSERT_TRUE(encoded.find(FRAME_BEGIN) != std::string::npos);
00010     TEST_ASSERT_TRUE(encoded.find(FRAME_END) != std::string::npos);
00011     TEST_ASSERT_TRUE(encoded.find("test_value") != std::string::npos);
00012 }
00013
00014 void test_frame_decode_basic() {
00015     Frame original = create_test_frame();
00016     std::string encoded = frame_encode(original);
00017     Frame decoded = frame_decode(encoded);
00018
00019     TEST_ASSERT_EQUAL(original.direction, decoded.direction);
00020     TEST_ASSERT_EQUAL(original.group, decoded.group);
00021     TEST_ASSERT_EQUAL(original.command, decoded.command);
00022     TEST_ASSERT_EQUAL_STRING(original.value.c_str(), decoded.value.c_str());
00023     TEST_ASSERT_EQUAL_STRING(original.unit.c_str(), decoded.unit.c_str());
00024 }
00025
00026 void test_frame_decode_invalid_header() {
00027     std::string invalid_frame = "INVALID" + std::string(1, DELIMITER) + "rest_of_frame";
00028     bool exceptionThrown = false;
00029
00030     try {
00031         Frame decoded = frame_decode(invalid_frame);
00032     } catch (const std::runtime_error& e) {
00033         exceptionThrown = true;
00034     } catch (...) {
00035         // Catch any other exceptions to avoid crashing the test
00036     }
00037
00038     TEST_ASSERT_TRUE(exceptionThrown);
00039 }
```

10.124 test/comms/test_frame_common.h File Reference

```
#include "unity.h"
#include "protocol.h"
```

```
#include "communication.h"
```

Include dependency graph for test_frame_common.h:

This graph shows which files directly or indirectly include this file:

Functions

- Frame create_test_frame ()

10.124.1 Function Documentation

10.124.1.1 create_test_frame()

```
Frame create_test_frame () [inline]
```

Definition at line 9 of file test_frame_common.h.

Here is the caller graph for this function:

10.125 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
```

10.126 test/comms/test_frame_send.cpp File Reference

```
#include "unity.h"
#include "communication.h"
#include "../mocks/hardware_mocks.h"
```

Include dependency graph for test_frame_send.cpp:

Functions

- void setUp (void)
- void tearDown (void)
- void test_send_frame_uart (void)

10.126.1 Function Documentation

10.126.1.1 setUp()

```
void setUp (
            void )
```

Definition at line 6 of file test_frame_send.cpp.

10.126.1.2 tearDown()

```
void tearDown (
            void )
```

Definition at line 12 of file test_frame_send.cpp.

10.126.1.3 test_send_frame_uart()

```
void test_send_frame_uart (
            void )
```

Definition at line 17 of file test_frame_send.cpp.

Here is the call graph for this function:

10.127 test_frame_send.cpp

Go to the documentation of this file.

```
00001 // test/comms/test_frame_send.cpp
00002 #include "unity.h"
00003 #include "communication.h"
00004 #include "../mocks/hardware_mocks.h"
00005
00006 void setUp(void) {
00007     // Enable mocks before each test
00008     mock_uart_enabled = true;
00009     uart_output_buffer.clear();
00010 }
00011
00012 void tearDown(void) {
00013     // Disable mocks after each test
00014     mock_uart_enabled = false;
00015 }
00016
00017 void test_send_frame_uart(void) {
00018     // Create a test frame
00019     Frame test_frame = {
00020         .operationType = OperationType::VAL,
00021         .group = 1,
00022         .command = 2,
00023         .value = "TEST_VALUE"
00024     };
00025
00026     // Call function under test
00027     send_frame_uart(test_frame);
00028
00029     // Verify output using mocks
00030     TEST_ASSERT_EQUAL(1, uart_output_buffer.size());
00031     TEST_ASSERT_TRUE(uart_output_buffer[0].find("KBST;0;VAL;1;2;TEST_VALUE;") != std::string::npos);
00032 }
```

10.128 test/mocks/hardware_mocks.cpp File Reference

```
#include "hardware_mocks.h"
#include <cstring>
```

Include dependency graph for hardware_mocks.cpp:

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

10.128.1 Function Documentation

10.128.1.1 mock_uart_puts()

```
void mock_uart_puts (
            uart_inst_t * uart,
            const char * str)
```

Definition at line 9 of file hardware_mocks.cpp.

10.128.1.2 mock_uart_init()

```
void mock_uart_init (
            uart_inst_t * uart,
            uint baudrate)
```

Definition at line 17 of file hardware_mocks.cpp.

10.128.1.3 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.

10.128.1.4 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.

10.128.2 Variable Documentation

10.128.2.1 mock_uart_enabled

```
bool mock_uart_enabled = false
```

Definition at line 6 of file hardware_mocks.cpp.

10.128.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer
```

Definition at line 7 of file hardware_mocks.cpp.

10.128.2.3 mock_spi_enabled

```
bool mock_spi_enabled = false
```

Definition at line 24 of file hardware_mocks.cpp.

10.128.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer
```

Definition at line 25 of file hardware_mocks.cpp.

10.129 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 }
```

10.130 test/mocks/hardware_mocks.h File Reference

```
#include <vector>
#include <string>
```

Include dependency graph for hardware_mocks.h:

This graph shows which files directly or indirectly include this file:

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

10.130.1 Function Documentation

10.130.1.1 mock_uart_puts()

```
void mock_uart_puts (
            uart_inst_t * uart,
            const char * str)
```

Definition at line 9 of file hardware_mocks.cpp.

10.130.1.2 mock_uart_init()

```
void mock_uart_init (
            uart_inst_t * uart,
            uint baudrate)
```

Definition at line 17 of file hardware_mocks.cpp.

10.130.1.3 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.

10.130.1.4 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.

10.130.2 Variable Documentation

10.130.2.1 mock_uart_enabled

```
bool mock_uart_enabled  [extern]
```

Definition at line 6 of file hardware_mocks.cpp.

10.130.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer  [extern]
```

Definition at line 7 of file hardware_mocks.cpp.

10.130.2.3 mock_spi_enabled

```
bool mock_spi_enabled  [extern]
```

Definition at line 24 of file hardware_mocks.cpp.

10.130.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer  [extern]
```

Definition at line 25 of file hardware_mocks.cpp.

10.131 hardware_mocks.h

Go to the documentation of this file.

```
00001 // test/mocks/hardware_mocks.h
00002 #ifndef HARDWARE_MOCKS_H
00003 #define HARDWARE_MOCKS_H
00004
00005 #include <vector>
00006 #include <string>
00007
00008 // UART mocks
00009 extern bool mock_uart_enabled;
00010 extern std::vector<std::string> uart_output_buffer;
00011
00012 void mock_uart_puts(uart_inst_t* uart, const char* str);
00013 void mock_uart_init(uart_inst_t* uart, uint baudrate);
00014
00015 // SPI mocks
00016 extern bool mock_spi_enabled;
00017 extern std::vector<uint8_t> spi_output_buffer;
00018
00019 void mock_spi_write_blocking(spi_inst_t* spi, const uint8_t* src, size_t len);
00020 int mock_spi_read_blocking(spi_inst_t* spi, uint8_t tx_data, uint8_t* dst, size_t len);
00021
00022 #endif // HARDWARE_MOCKS_H
```

10.132 test/test_mocks.cpp File Reference

```
#include "utils.h"
#include "protocol.h"
#include "PowerManager.h"
#include "DS3231.h"
#include "lib/location/NMEA/NMEA_data.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <map>
#include <cstdio>
```

Include dependency graph for test_mocks.cpp:

Functions

- void uart_print (const std::string &msg, VerbosityLevel level, bool add_timestamp, uart_inst_t *uart)
- std::string get_last_telemetry_record_csv ()
- std::string get_last_sensor_record_csv ()
- void test_error_code_conversion ()
- void test_command_handler_get_operation ()
- void test_command_handler_set_operation ()
- void test_command_handler_invalid_operation ()

Variables

- PowerManager powerManager (nullptr)
- DS3231 systemClock (nullptr)
- NMEAData nmea_data
- volatile bool g_pending_bootloader_reset = false
- VerbosityLevel g_uart_verbosity = VerbosityLevel::DEBUG
- std::string uart_output_buffer
- bool mock_uart_enabled = false

10.132.1 Function Documentation

10.132.1.1 uart_print()

```
void uart_print (
            const std::string & msg,
            VerbosityLevel level,
            bool add_timestamp,
            uart_inst_t * uart)
```

Definition at line 26 of file test_mocks.cpp.

10.132.1.2 get_last_telemetry_record_csv()

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

Definition at line 89 of file test_mocks.cpp.

10.132.1.3 get_last_sensor_record_csv()

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

Definition at line 93 of file test_mocks.cpp.

10.132.1.4 test_error_code_conversion()

```
void test_error_code_conversion (
            void )
```

Definition at line 101 of file test_mocks.cpp.

Here is the caller graph for this function:

10.132.1.5 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
            void )
```

Definition at line 102 of file test_mocks.cpp.

10.132.1.6 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
            void )
```

Definition at line 103 of file test_mocks.cpp.

10.132.1.7 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
            void )
```

Definition at line 104 of file test_mocks.cpp.

10.132.2 Variable Documentation

10.132.2.1 powerManager

```
PowerManager powerManager(nullptr) (
            nullptr )
```

10.132.2.2 systemClock

```
DS3231 systemClock(nullptr) (
            nullptr )
```

10.132.2.3 nmea_data

```
NMEAData nmea_data
```

Definition at line 17 of file test_mocks.cpp.

10.132.2.4 g_pending_bootloader_reset

```
volatile bool g_pending_bootloader_reset = false
```

Definition at line 18 of file test_mocks.cpp.

10.132.2.5 g_uart_verbosity

```
VerbosityLevel g_uart_verbosity = VerbosityLevel::DEBUG
```

Definition at line 19 of file test_mocks.cpp.

10.132.2.6 uart_output_buffer

```
std::string uart_output_buffer
```

Definition at line 22 of file test_mocks.cpp.

10.132.2.7 mock_uart_enabled

```
bool mock_uart_enabled = false
```

Definition at line 23 of file test_mocks.cpp.

10.133 test_mocks.cpp

Go to the documentation of this file.

```
// test/test_mocks.cpp - Mock implementations for unit tests

#include "utils.h"
#include "protocol.h"
#include "PowerManager.h"
#include "DS3231.h"
#include "lib/location/NMEA/NMEA_data.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <map>
#include <cstdio>

// Mock global variables
PowerManager powerManager(nullptr);
DS3231 systemClock(nullptr);
NMEAData nmea_data;
volatile bool g_pending_bootloader_reset = false;
VerbosityLevel g_uart_verbosity = VerbosityLevel::DEBUG;

// Mock UART buffers for testing
std::string uart_output_buffer;
bool mock_uart_enabled = false;

// Mock UART print function
void uart_print(const std::string& msg, VerbosityLevel level, bool add_timestamp, uart_inst_t* uart) {
    if (mock_uart_enabled) {
        uart_output_buffer += msg + "\n";
    }
    // In test environment, we don't actually print to UART
}

// NMEAData methods
NMEAData::NMEAData() {}

std::vector<std::string> NMEAData::get_rmc_tokens() const {
    return {"$GPRMC", "123519", "A", "4807.038", "N", "01131.000", "E", "022.4", "084.4", "230394", "003.1", "W"};
}

std::vector<std::string> NMEAData::get_gga_tokens() const {
    return {"$GPGGA", "123519", "4807.038", "N", "01131.000", "E", "1", "08", "0.9", "545.4", "M", "46.9", "M", "", ""};
}

bool NMEAData::has_valid_time() const {
    return true;
}

time_t NMEAData::get_unix_time() const {
    return 1234567890; // Some fixed timestamp for tests
}

// Implementation of SensorWrapper singleton methods - not redefining the class itself
// This creates concrete implementations of the methods declared in ISensor.h
SensorWrapper& SensorWrapper::get_instance() {
    static SensorWrapper instance;
    return instance;
}

bool SensorWrapper::sensor_init(SensorType type, i2c_inst_t* i2c) {
    return true;  // Always succeed in tests
}

bool SensorWrapper::sensor_configure(SensorType type, const std::map<std::string, std::string>& config) {
    return true;  // Always succeed in tests
}

float SensorWrapper::sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType) {
    return 42.0f;  // Return fixed test value
}

ISensor* SensorWrapper::get_sensor(SensorType type) {
    return nullptr;  // For tests, we don't need to return actual sensor instances
}

std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::scan_connected_sensors(i2c_inst_t* i2c) {
    return {
        {SensorType::ENVIRONMENT, 0x76},
        {SensorType::LIGHT, 0x23}
    };
```

```
}

std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::get_available_sensors() {
    return {
        {SensorType::ENVIRONMENT, 0x76},
        {SensorType::LIGHT, 0x23}
    };
}

std::string get_last_telemetry_record_csv() {
    return "mock_timestamp,mock_value1,mock_value2";
}

std::string get_last_sensor_record_csv() {
    return "mock_timestamp,mock_sensor1,mock_sensor2";
}

// Private constructor implementation (that would be defined in ISensor.cpp)
SensorWrapper::SensorWrapper() {}

// Mock test cases that were referenced but not defined
void test_error_code_conversion() {}
void test_command_handler_get_operation() {}
void test_command_handler_set_operation() {}
void test_command_handler_invalid_operation() {}
```

10.134 test/test_runner.cpp File Reference

```
#include "includes.h"
#include "unity.h"
```

Include dependency graph for test_runner.cpp:

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)

10.134.1 Function Documentation

10.134.1.1 test_frame_encode_basic()

```
void test_frame_encode_basic (
            void )  [extern]
```

Definition at line 4 of file test_frame_coding.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
            void )  [extern]
```

Definition at line 14 of file test_frame_coding.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
            void )  [extern]
```

Definition at line 26 of file test_frame_coding.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.4 test_frame_build_get()

```
void test_frame_build_get (
            void )  [extern]
```

Definition at line 24 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.5 test_frame_build_set()

```
void test_frame_build_set (
            void )  [extern]
```

Definition at line 35 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.6 test_frame_build_res()

```
void test_frame_build_res (
            void )  [extern]
```

Definition at line 46 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.7 test_frame_build_seq()

```
void test_frame_build_seq (
            void )  [extern]
```

Definition at line 57 of file test_frame_build.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.8 test_frame_build_val()

```
void test_frame_build_val (
            void )  [extern]
```

Definition at line 4 of file test_frame_build.cpp.

Here is the call graph for this function:

10.134.1.9 test_frame_build_err()

```
void test_frame_build_err (
            void )  [extern]
```

Definition at line 15 of file test_frame_build.cpp.

Here is the call graph for this function:

10.134.1.10 test_operation_type_conversion()

```
void test_operation_type_conversion (
            void )  [extern]
```

Definition at line 4 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.11 test_value_unit_type_conversion()

```
void test_value_unit_type_conversion (
            void )  [extern]
```

Definition at line 13 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.12 test_exception_type_conversion()

```
void test_exception_type_conversion (
            void )  [extern]
```

Definition at line 20 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.13 test_hex_string_conversion()

```
void test_hex_string_conversion (
            void )  [extern]
```

Definition at line 27 of file test_converters.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.14 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
            void )  [extern]
```

Definition at line 29 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.15 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
            void )  [extern]
```

Definition at line 39 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.16 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
            void )  [extern]
```

Definition at line 54 of file test_comand_handlers.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.17 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
            void )  [extern]
```

Definition at line 6 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.18 test_handle_get_build_version()

```
void test_handle_get_build_version (
            void )  [extern]
```

Definition at line 15 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.19 test_handle_verbosity()

```
void test_handle_verbosity (
            void )  [extern]
```

Definition at line 25 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.20 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
            void )  [extern]
```

Definition at line 40 of file test_diagnostic_commands.cpp.

Here is the call graph for this function:

Here is the caller graph for this function:

10.134.1.21 test_error_code_conversion()

```
void test_error_code_conversion (
            void )  [extern]
```

Definition at line 101 of file test_mocks.cpp.

Here is the caller graph for this function:

10.134.1.22 main()

```
int main (
            void )
```

Definition at line 38 of file test_runner.cpp.

Here is the call graph for this function:

10.135 test_runner.cpp

Go to the documentation of this file.

```
00001 // test/test_runner.cpp
00002 #include "includes.h"
00003 #include "unity.h"
00004
00005 // External test function declarations
00006 // Pure software tests (no hardware dependencies)
00007 extern void test_frame_encode_basic(void);
00008 extern void test_frame_decode_basic(void);
00009 extern void test_frame_decode_invalid_header(void);
00010
00011 // FRAME BUILD
00012 extern void test_frame_build_get(void);
00013 extern void test_frame_build_set(void);
00014 extern void test_frame_build_res(void);
00015 extern void test_frame_build_seq(void);
00016 extern void test_frame_build_val(void);
00017 extern void test_frame_build_err(void);
00018
00019 extern void test_operation_type_conversion(void);
00020 extern void test_value_unit_type_conversion(void);
00021 extern void test_exception_type_conversion(void);
00022 extern void test_hex_string_conversion(void);
00023
00024 // Command handler tests
00025 extern void test_command_handler_get_operation(void);
00026 extern void test_command_handler_set_operation(void);
00027 extern void test_command_handler_invalid_operation(void);
00028
00029 //diagnostic
00030 extern void test_handle_get_commands_list(void);
00031 extern void test_handle_get_build_version(void);
00032 extern void test_handle_verbosity(void);
00033 extern void test_handle_enter_bootloader_mode(void);
00034
```

```
// 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();
}
```

10.136 todo.md File Reference

Index

_BH1750_DEFAULT_MTREG
BH1750.h, 325
_BH1750_DEVICE_ID
BH1750.h, 325
_BH1750_MTREG_MAX
BH1750.h, 325
_BH1750_MTREG_MIN
BH1750.h, 325
__attribute__
Event Management, 96, 99
_filterRes
INA3221, 192
_i2c
INA3221, 192
_i2c_addr
BH1750, 148
INA3221, 192
_masken_reg
INA3221, 192
_read
INA3221, 189
_shuntRes
INA3221, 192
_write
INA3221, 190
~ISensor
ISensor, 194

ADDR_SDO_HIGH
BME280, 156
ADDR_SDO_LOW
BME280, 156
altitude
TelemetryRecord, 225
ANSI_BLUE
utils.h, 362
ANSI_GREEN
utils.h, 362
ANSI_RED
utils.h, 362
ANSI_RESET
utils.h, 362
ANSI_YELLOW
utils.h, 362
avg_mode
INA3221::conf_reg_t, 171

BATTERY_FULL
Event Management, 95
BATTERY_FULL_THRESHOLD
PowerManager, 208
BATTERY_LOW
Event Management, 95
BATTERY_LOW_THRESHOLD
PowerManager, 208
BATTERY_NORMAL
Event Management, 95
battery_voltage
TelemetryRecord, 223
bcd_to_bin
RTC clock, 91
begin
BH1750, 146
Configuration Functions, 103
BH1750, 145
_i2c_addr, 148
begin, 146
BH1750, 146
configure, 146
CONTINUOUS_HIGH_RES_MODE, 146
CONTINUOUS_HIGH_RES_MODE_2, 146
CONTINUOUS_LOW_RES_MODE, 146
get_light_level, 147
i2c_port_, 148
Mode, 145
ONE_TIME_HIGH_RES_MODE, 146
ONE_TIME_HIGH_RES_MODE_2, 146
ONE_TIME_LOW_RES_MODE, 146
POWER_ON, 146
RESET, 146
UNCONFIGURED_POWER_DOWN, 146
write8, 147
BH1750.h
_BH1750_DEFAULT_MTREG, 325
_BH1750_DEVICE_ID, 325
_BH1750_MTREG_MAX, 325
_BH1750_MTREG_MIN, 325
BH1750Wrapper, 148
BH1750Wrapper, 149
configure, 151
get_address, 151
get_i2c_addr, 150
get_type, 151
i2c_port_, 152
init, 150
initialized_, 152
is_initialized, 150
read_data, 150
sensor_, 152

bin_to_bcd
RTC clock, 89
BME280, 153
ADDR_SDO_HIGH, 156
ADDR_SDO_LOW, 156
BME280, 154
calib_params, 157
configure_sensor, 156
convert_humidity, 155
convert_pressure, 155
convert_temperature, 155
device_addr, 157
get_calibration_parameters, 156
i2c_port, 157
init, 155
initialized_, 157
NUM_CALIB_PARAMS, 162
read_raw_all, 155
REG_CONFIG, 157
REG_CTRL_HUM, 157
REG_CTRL_MEAS, 157
REG_DIG_H1, 161
REG_DIG_H2, 161
REG_DIG_H3, 161
REG_DIG_H4, 161
REG_DIG_H5, 162
REG_DIG_H6, 162
REG_DIG_P1_LSB, 159
REG_DIG_P1_MSB, 159
REG_DIG_P2_LSB, 159
REG_DIG_P2_MSB, 159
REG_DIG_P3_LSB, 159
REG_DIG_P3_MSB, 159
REG_DIG_P4_LSB, 160
REG_DIG_P4_MSB, 160
REG_DIG_P5_LSB, 160
REG_DIG_P5_MSB, 160
REG_DIG_P6_LSB, 160
REG_DIG_P6_MSB, 160
REG_DIG_P7_LSB, 160
REG_DIG_P7_MSB, 160
REG_DIG_P8_LSB, 161
REG_DIG_P8_MSB, 161
REG_DIG_P9_LSB, 161
REG_DIG_P9_MSB, 161
REG_DIG_T1_LSB, 158
REG_DIG_T1_MSB, 158
REG_DIG_T2_LSB, 158
REG_DIG_T2_MSB, 158
REG_DIG_T3_LSB, 159
REG_DIG_T3_MSB, 159
REG_HUMIDITY_MSB, 158
REG_PRESSURE_MSB, 158
REG_RESET, 158
REG_TEMPERATURE_MSB, 158
reset, 155
t_fine, 157
BME280CalibParam, 162
dig_h1, 164
dig_h2, 164
dig_h3, 164
dig_h4, 165
dig_h5, 165
dig_h6, 165
dig_p1, 163
dig_p2, 163
dig_p3, 163
dig_p4, 163
dig_p5, 164
dig_p6, 164
dig_p7, 164
dig_p8, 164
dig_p9, 164
dig_t1, 163
dig_t2, 163
dig_t3, 163
BME280Wrapper, 165
BME280Wrapper, 167
configure, 168
get_address, 168
get_type, 168
init, 167
initialized_, 169
is_initialized, 167
read_data, 167
sensor_, 169
BOOL
Protocol, 63
BOOT
Event Management, 95
buffer
main.cpp, 368
buffer_index
main.cpp, 368
BUFFER_SIZE
pin_config.h, 304
BUILD_NUMBER
build_number.h, 227
build_number.h, 227
BUILD_NUMBER, 227
build_version
TelemetryRecord, 222
bus_conv_time
INA3221::conf_reg_t, 170

calib_params
BME280, 157
century
ds3231_data_t, 177
ch1_en
INA3221::conf_reg_t, 171
ch2_en
INA3221::conf_reg_t, 171
ch3_en
INA3221::conf_reg_t, 171
CHANGED
Event Management, 96

charge_current_solar
TelemetryRecord, 223
charge_current_usb
TelemetryRecord, 223
CHARGE_SOLAR
power_commands.cpp, 257
CHARGE_TOTAL
power_commands.cpp, 257
CHARGE_USB
power_commands.cpp, 256
charging_solar_active_
PowerManager, 209
charging_usb_active_
PowerManager, 209
CLOCK
Event Management, 94
Clock Commands, 13
Clock Management Commands, 23
handle_clock_sync_interval, 25
handle_get_last_sync_time, 26
handle_time, 23
handle_timezone_offset, 24
clock_commands.cpp
CLOCK_GROUP, 240
CLOCK_SYNC_INTERVAL, 240
LAST_SYNC_TIME, 240
TIME, 240
TIMEZONE_OFFSET, 240
clock_enable
RTC clock, 79
CLOCK_GROUP
clock_commands.cpp, 240
clock_mutex_
DS3231, 175
CLOCK_SYNC_INTERVAL
clock_commands.cpp, 240
ClockEvent
Event Management, 96
collect_gps_data
Location, 100
collect_gps_telemetry
Telemetry Manager, 133
collect_power_telemetry
Telemetry Manager, 131
collect_sensor_telemetry
Telemetry Manager, 134
collect_telemetry
Telemetry Manager, 134
command
Frame, 186
Command System, 27
command_handlers, 28
CommandHandler, 27
CommandMap, 27
execute_command, 27
command_handlers
Command System, 28
CommandAccessLevel
Protocol, 62
CommandHandler
Command System, 27
frame.cpp, 277
CommandMap
Command System, 27
COMMS
Event Management, 94
CommsEvent
Event Management, 95
communication.cpp
initialize_radio, 269
interval, 271
lastPrintTime, 271
lastReceiveTime, 271
lastSendTime, 271
lora_tx_done_callback, 270
msgCount, 271
outgoing, 271
communication.h
determine_unit, 276
initialize_radio, 273
lora_tx_done_callback, 274
send_frame_lora, 275
send_frame_uart, 274
send_message, 274
split_and_send_message, 276
Configuration Functions, 102
begin, 103
get_die_id, 105
get_manufacturer_id, 104
INA3221, 103
read_register, 106
reset, 104
set_averaging_mode, 110
set_bus_conversion_time, 110
set_bus_measurement_disable, 109
set_bus_measurement_enable, 109
set_mode_continuous, 107
set_mode_power_down, 106
set_mode_triggered, 107
set_shunt_conversion_time, 111
set_shunt_measurement_disable, 108
set_shunt_measurement_enable, 108
configure
BH1750, 146
BH1750Wrapper, 151
BME280Wrapper, 168
ISensor, 195
Power Management, 120
configure_sensor
BME280, 156
CONTINUOUS_HIGH_RES_MODE
BH1750, 146
CONTINUOUS_HIGH_RES_MODE_2
BH1750, 146
CONTINUOUS_LOW_RES_MODE
BH1750, 146

conv_ready
INA3221::masken_reg_t, 196
convert_humidity
BME280, 155
convert_pressure
BME280, 155
convert_temperature
BME280, 155
core1_entry
main.cpp, 366
CORE1_START
Event Management, 95
CORE1_STOP
Event Management, 95
course
TelemetryRecord, 224
create_test_frame
test_frame_common.h, 393
crit_alert_ch1
INA3221::masken_reg_t, 197
crit_alert_ch2
INA3221::masken_reg_t, 197
crit_alert_ch3
INA3221::masken_reg_t, 197
crit_alert_latch_en
INA3221::masken_reg_t, 197

DATA_READY
Event Management, 96
date
ds3231_data_t, 177
TelemetryRecord, 225
DATETIME
Protocol, 63
day
ds3231_data_t, 177
days_of_week
DS3231.h, 237
DEBUG
utils.h, 363
DEBUG_UART_BAUD_RATE
pin_config.h, 302
DEBUG_UART_PORT
pin_config.h, 302
DEBUG_UART_RX_PIN
pin_config.h, 302
DEBUG_UART_TX_PIN
pin_config.h, 302
DEFAULT_FLUSH_THRESHOLD
Telemetry Manager, 131
DEFAULT_SAMPLE_INTERVAL_MS
Telemetry Manager, 130
DELIMITER
protocol.h, 281
determine_unit
communication.h, 276
device_addr
BME280, 157
Diagnostic Commands, 29
handle_enter_bootloader_mode, 32
handle_get_build_version, 30
handle_get_commands_list, 29
handle_verbosity, 31
dig_h1
BME280CalibParam, 164
dig_h2
BME280CalibParam, 164
dig_h3
BME280CalibParam, 164
dig_h4
BME280CalibParam, 165
dig_h5
BME280CalibParam, 165
dig_h6
BME280CalibParam, 165
dig_p1
BME280CalibParam, 163
dig_p2
BME280CalibParam, 163
dig_p3
BME280CalibParam, 163
dig_p4
BME280CalibParam, 163
dig_p5
BME280CalibParam, 164
dig_p6
BME280CalibParam, 164
dig_p7
BME280CalibParam, 164
dig_p8
BME280CalibParam, 164
dig_p9
BME280CalibParam, 164
dig_t1
BME280CalibParam, 163
dig_t2
BME280CalibParam, 163
dig_t3
BME280CalibParam, 163
direction
Frame, 186
discharge_current
TelemetryRecord, 223
DRAW_TOTAL
power_commands.cpp, 257
DS3231, 171
clock_mutex_, 175
DS3231, 173, 174
ds3231_addr, 175
i2c, 175
last_sync_time_, 175
operator=, 174
RTC clock, 72
sync_interval_minutes_, 175
timezone_offset_minutes_, 175
DS3231.h
days_of_week, 237

DS3231_CONTROL_REG, 236
DS3231_CONTROL_STATUS_REG, 237
DS3231_DATE_REG, 236
DS3231_DAY_REG, 236
DS3231_DEVICE_ADRESS, 235
DS3231_HOURS_REG, 236
DS3231_MINUTES_REG, 235
DS3231_MONTH_REG, 236
DS3231_SECONDS_REG, 235
DS3231_TEMPERATURE_LSB_REG, 237
DS3231_TEMPERATURE_MSB_REG, 237
DS3231_YEAR_REG, 236
FRIDAY, 237
MONDAY, 237
SATURDAY, 237
SUNDAY, 237
THURSDAY, 237
TUESDAY, 237
WEDNESDAY, 237
ds3231_addr
DS3231, 175
DS3231_CONTROL_REG
DS3231.h, 236
DS3231_CONTROL_STATUS_REG
DS3231.h, 237
ds3231_data_t, 176
century, 177
date, 177
day, 177
hours, 176
minutes, 176
month, 177
seconds, 176
year, 177
DS3231_DATE_REG
DS3231.h, 236
DS3231_DAY_REG
DS3231.h, 236
DS3231_DEVICE_ADRESS
DS3231.h, 235
DS3231_HOURS_REG
DS3231.h, 236
DS3231_MINUTES_REG
DS3231.h, 235
DS3231_MONTH_REG
DS3231.h, 236
DS3231_SECONDS_REG
DS3231.h, 235
DS3231_TEMPERATURE_LSB_REG
DS3231.h, 237
DS3231_TEMPERATURE_MSB_REG
DS3231.h, 237
DS3231_YEAR_REG
DS3231.h, 236

emit
EventEmitter, 178
emit_power_events
Telemetry Manager, 132
ENVIRONMENT
Sensors, 123
ERR
Protocol, 62
ERROR
Event Management, 96
utils.h, 363
error_code_to_string
Utility Converters, 68
ErrorCode
Protocol, 61
event
event_manager.h, 293
EventLog, 180
Event Commands, 34
handle_get_event_count, 36
handle_get_last_events, 35
Event Management, 92
__attribute__, 96, 99
BATTERY_FULL, 95
BATTERY_LOW, 95
BATTERY_NORMAL, 95
BOOT, 95
CHANGED, 96
CLOCK, 94
ClockEvent, 96
COMMS, 94
CommsEvent, 95
CORE1_START, 95
CORE1_STOP, 95
DATA_READY, 96
ERROR, 96
EVENT_BUFFER_SIZE, 94
EVENT_FLUSH_THRESHOLD, 94
EVENT_LOG_FILE, 94
EventGroup, 94
get_event, 98
GPS, 94
GPS_SYNC, 96
GPS_SYNC_DATA_NOT_READY, 96
GPSEvent, 95
init, 96
LOCK, 96
log_event, 97
LOST, 96
MSG_RECEIVED, 95
MSG_SENT, 95
PASS_THROUGH_END, 96
PASS_THROUGH_START, 96
POWER, 94
POWER_FALLING, 95
POWER_OFF, 96
POWER_ON, 96
PowerEvent, 95
RADIO_ERROR, 95
RADIO_INIT, 95
save_to_storage, 98
SHUTDOWN, 95

SOLAR_ACTIVE, 95
SOLAR_INACTIVE, 95
SYSTEM, 94
SystemEvent, 94
UART_ERROR, 95
USB_CONNECTED, 95
USB_DISCONNECTED, 95
WATCHDOG_RESET, 95
EVENT_BUFFER_SIZE
Event Management, 94
EVENT_FLUSH_THRESHOLD
Event Management, 94
EVENT_LOG_FILE
Event Management, 94
event_manager.h
event, 293
group, 293
id, 293
timestamp, 293
eventCount
EventManager, 184
EventEmitter, 178
emit, 178
EventGroup
Event Management, 94
EventLog, 179
event, 180
group, 180
id, 180
timestamp, 180
EventManager, 180
eventCount, 184
EventManager, 182
eventMutex, 184
events, 184
eventsSinceFlush, 185
get_event_count, 183
get_instance, 182
load_from_storage, 183
nextEventId, 184
operator=, 182
writeIndex, 184
eventMutex
EventManager, 184
eventRegister
frame.cpp, 278
events
EventManager, 184
eventsSinceFlush
EventManager, 185
exception_type_to_string
Utility Converters, 66
ExceptionType
Protocol, 63
execute_command
Command System, 27

FAIL_TO_SET
Protocol, 62
fix_quality
TelemetryRecord, 225
flush_sensor_data
Telemetry Manager, 136
flush_telemetry
Telemetry Manager, 135
flush_threshold
Telemetry Manager, 140
footer
Frame, 186
Frame, 185
command, 186
direction, 186
footer, 186
group, 186
header, 186
operationType, 186
unit, 186
value, 186
Frame Handling, 55
frame_build, 59
frame_decode, 57
frame_encode, 56
frame_process, 58
frame.cpp
CommandHandler, 277
eventRegister, 278
FRAME_BEGIN
protocol.h, 281
frame_build
Frame Handling, 59
frame_decode
Frame Handling, 57
frame_encode
Frame Handling, 56
FRAME_END
protocol.h, 281
frame_process
Frame Handling, 58
FRIDAY
DS3231.h, 237
fs_init
Storage, 127
fs_stop
Storage, 127

g_pending_bootloader_reset
test_mocks.cpp, 404
g_uart_verbosity
test_mocks.cpp, 404
GET
Protocol, 62
get_address
BH1750Wrapper, 151
BME280Wrapper, 168
ISensor, 195
get_available_sensors
Sensors, 125
get_calibration_parameters

BME280, 156
get_clock_sync_interval
RTC clock, 81
get_current
INA3221, 191
get_current_charge_solar
Power Management, 119
get_current_charge_total
Power Management, 119
get_current_charge_usb
Power Management, 118
get_current_draw
Power Management, 118
get_current_ma
Measurement Functions, 113
get_die_id
Configuration Functions, 105
get_event
Event Management, 98
get_event_count
EventManager, 183
get_gga_tokens
NMEAData, 202
get_i2c_addr
BH1750Wrapper, 150
get_instance
EventManager, 182
NMEAData, 200
Power Management, 115
RTC clock, 72
SensorWrapper, 212
SystemStateManager, 216
GET_LAST_SENSOR_RECORD_COMMAND
telemetry_commands.cpp, 267
get_last_sensor_record_csv
Telemetry Manager, 139
test_mocks.cpp, 402
get_last_sync_time
RTC clock, 83
GET_LAST_TELEMETRY_RECORD_COMMAND
telemetry_commands.cpp, 267
get_last_telemetry_record_csv
Telemetry Manager, 138
test_mocks.cpp, 402
get_level_color
utils.cpp, 356
get_level_prefix
utils.cpp, 357
get_light_level
BH1750, 147
get_local_time
RTC clock, 84
get_manufacturer_id
Configuration Functions, 104
get_rmc_tokens
NMEAData, 202
get_sensor
Sensors, 124
get_shunt_voltage
Measurement Functions, 112
get_time
RTC clock, 74
get_timezone_offset
RTC clock, 80
get_type
BH1750Wrapper, 151
BME280Wrapper, 168
ISensor, 194
get_uart_verbosity
SystemStateManager, 219
get_unix_time
NMEAData, 203
RTC clock, 78
get_voltage
Measurement Functions, 113
get_voltage_5v
Power Management, 118
get_voltage_battery
Power Management, 117
GGA_DATA_COMMAND
gps_commands.cpp, 252
gga_mutex_
NMEAData, 204
gga_tokens_
NMEAData, 203
GPS
Event Management, 94
GPS Commands, 37
handle_enable_gps_uart_passthrough, 38
handle_get_gga_data, 40
handle_get_rmc_data, 39
handle_gps_power_status, 37
gps_collection_paused
SystemStateManager, 221
gps_commands.cpp
GGA_DATA_COMMAND, 252
GPS_GROUP, 252
PASSTHROUGH_COMMAND, 252
POWER_STATUS_COMMAND, 252
RMC_DATA_COMMAND, 252
GPS_GROUP
gps_commands.cpp, 252
GPS_POWER_ENABLE_PIN
pin_config.h, 303
GPS_SYNC
Event Management, 96
GPS_SYNC_DATA_NOT_READY
Event Management, 96
GPS_UART_BAUD_RATE
pin_config.h, 303
GPS_UART_PORT
pin_config.h, 303
GPS_UART_RX_PIN
pin_config.h, 303
GPS_UART_TX_PIN
pin_config.h, 303

GPSEvent
Event Management, 95
group
event_manager.h, 293
EventLog, 180
Frame, 186

handle_clock_sync_interval
Clock Management Commands, 25
handle_enable_gps_uart_passthrough
GPS Commands, 38
handle_enter_bootloader_mode
Diagnostic Commands, 32
handle_get_build_version
Diagnostic Commands, 30
handle_get_commands_list
Diagnostic Commands, 29
handle_get_current_charge_solar
Power Commands, 45
handle_get_current_charge_total
Power Commands, 46
handle_get_current_charge_usb
Power Commands, 44
handle_get_current_draw
Power Commands, 47
handle_get_event_count
Event Commands, 36
handle_get_gga_data
GPS Commands, 40
handle_get_last_events
Event Commands, 35
handle_get_last_sensor_record
Telemetry Buffer Commands, 54
handle_get_last_sync_time
Clock Management Commands, 26
handle_get_last_telemetry_record
Telemetry Buffer Commands, 54
handle_get_power_manager_ids
Power Commands, 41
handle_get_rmc_data
GPS Commands, 39
handle_get_sensor_data
Sensor Commands, 48
handle_get_sensor_list
Sensor Commands, 50
handle_get_voltage_5v
Power Commands, 43
handle_get_voltage_battery
Power Commands, 42
handle_gps_power_status
GPS Commands, 37
handle_list_files
Storage Commands, 52
handle_mount
Storage Commands, 52
handle_sensor_config
Sensor Commands, 49
handle_time
Clock Management Commands, 23
handle_timezone_offset
Clock Management Commands, 24
handle_uart_input
Receiving Data, 64
handle_verbosity
Diagnostic Commands, 31
hardware_mocks.cpp
mock_spi_enabled, 397
mock_spi_read_blocking, 397
mock_spi_write_blocking, 397
mock_uart_enabled, 397
mock_uart_init, 396
mock_uart_puts, 396
spi_output_buffer, 397
uart_output_buffer, 397
hardware_mocks.h
mock_spi_enabled, 400
mock_spi_read_blocking, 400
mock_spi_write_blocking, 400
mock_uart_enabled, 400
mock_uart_init, 400
mock_uart_puts, 400
spi_output_buffer, 401
uart_output_buffer, 400
has_valid_time
NMEAData, 202
header
Frame, 186
hex_string_to_bytes
Utility Converters, 70
hours
ds3231_data_t, 176
HUMIDITY
Sensors, 123
humidity
SensorDataRecord, 210

i2c
DS3231, 175
i2c_port
BME280, 157
i2c_port_
BH1750, 148
BH1750Wrapper, 152
i2c_read_reg
RTC clock, 87
i2c_write_reg
RTC clock, 88
id
event_manager.h, 293
EventLog, 180
INA3221, 187
_filterRes, 192
_i2c, 192
_i2c_addr, 192
_masken_reg, 192
_read, 189
_shuntRes, 192
_write, 190

Configuration Functions, 103
get_current, 191
INA3221 Power Monitor, 102
INA3221.h
INA3221_ADDR40_GND, 314
INA3221_ADDR41_VCC, 314
INA3221_ADDR42_SDA, 314
INA3221_ADDR43_SCL, 314
ina3221_addr_t, 313
ina3221_avg_mode_t, 315
INA3221_CH1, 314
INA3221_CH2, 314
INA3221_CH3, 314
INA3221_CH_NUM, 315
ina3221_ch_t, 314
ina3221_conv_time_t, 314
INA3221_REG_CH1_BUSV, 314
INA3221_REG_CH1_CRIT_ALERT_LIM, 314
INA3221_REG_CH1_SHUNTV, 314
INA3221_REG_CH1_WARNING_ALERT_LIM, 314
INA3221_REG_CH2_BUSV, 314
INA3221_REG_CH2_CRIT_ALERT_LIM, 314
INA3221_REG_CH2_SHUNTV, 314
INA3221_REG_CH2_WARNING_ALERT_LIM, 314
INA3221_REG_CH3_BUSV, 314
INA3221_REG_CH3_CRIT_ALERT_LIM, 314
INA3221_REG_CH3_SHUNTV, 314
INA3221_REG_CH3_WARNING_ALERT_LIM, 314
INA3221_REG_CONF, 314
INA3221_REG_CONF_AVG_1, 315
INA3221_REG_CONF_AVG_1024, 315
INA3221_REG_CONF_AVG_128, 315
INA3221_REG_CONF_AVG_16, 315
INA3221_REG_CONF_AVG_256, 315
INA3221_REG_CONF_AVG_4, 315
INA3221_REG_CONF_AVG_512, 315
INA3221_REG_CONF_AVG_64, 315
INA3221_REG_CONF_CT_1100US, 315
INA3221_REG_CONF_CT_140US, 315
INA3221_REG_CONF_CT_204US, 315
INA3221_REG_CONF_CT_2116US, 315
INA3221_REG_CONF_CT_332US, 315
INA3221_REG_CONF_CT_4156US, 315
INA3221_REG_CONF_CT_588US, 315
INA3221_REG_CONF_CT_8244US, 315
INA3221_REG_DIE_ID, 314
INA3221_REG_MANUF_ID, 314
INA3221_REG_MASK_ENABLE, 314
INA3221_REG_PWR_VALID_HI_LIM, 314
INA3221_REG_PWR_VALID_LO_LIM, 314
INA3221_REG_SHUNTV_SUM, 314
INA3221_REG_SHUNTV_SUM_LIM, 314
ina3221_reg_t, 314
SHUNT_VOLTAGE_LSB_UV, 315
INA3221::conf_reg_t, 169
avg_mode, 171
bus_conv_time, 170
ch1_en, 171
ch2_en, 171
ch3_en, 171
mode_bus_en, 170
mode_continious_en, 170
mode_shunt_en, 170
reset, 171
shunt_conv_time, 170
INA3221::masken_reg_t, 195
conv_ready, 196
crit_alert_ch1, 197
crit_alert_ch2, 197
crit_alert_ch3, 197
crit_alert_latch_en, 197
pwr_valid_alert, 196
reserved, 198
shunt_sum_alert, 197
shunt_sum_en_ch1, 198
shunt_sum_en_ch2, 198
shunt_sum_en_ch3, 197
timing_ctrl_alert, 196
warn_alert_ch1, 197
warn_alert_ch2, 196
warn_alert_ch3, 196
warn_alert_latch_en, 197
ina3221_
PowerManager, 208
INA3221_ADDR40_GND
INA3221.h, 314
INA3221_ADDR41_VCC
INA3221.h, 314
INA3221_ADDR42_SDA
INA3221.h, 314
INA3221_ADDR43_SCL
INA3221.h, 314
ina3221_addr_t
INA3221.h, 313
ina3221_avg_mode_t
INA3221.h, 315
INA3221_CH1
INA3221.h, 314
INA3221_CH2
INA3221.h, 314
INA3221_CH3
INA3221.h, 314
INA3221_CH_NUM
INA3221.h, 315
ina3221_ch_t
INA3221.h, 314
ina3221_conv_time_t
INA3221.h, 314
INA3221_REG_CH1_BUSV
INA3221.h, 314
INA3221_REG_CH1_CRIT_ALERT_LIM
INA3221.h, 314
INA3221_REG_CH1_SHUNTV

INA3221.h, 314
INA3221_REG_CH1_WARNING_ALERT_LIM
INA3221.h, 314
INA3221_REG_CH2_BUSV
INA3221.h, 314
INA3221_REG_CH2_CRIT_ALERT_LIM
INA3221.h, 314
INA3221_REG_CH2_SHUNTV
INA3221.h, 314
INA3221_REG_CH2_WARNING_ALERT_LIM
INA3221.h, 314
INA3221_REG_CH3_BUSV
INA3221.h, 314
INA3221_REG_CH3_CRIT_ALERT_LIM
INA3221.h, 314
INA3221_REG_CH3_SHUNTV
INA3221.h, 314
INA3221_REG_CH3_WARNING_ALERT_LIM
INA3221.h, 314
INA3221_REG_CONF
INA3221.h, 314
INA3221_REG_CONF_AVG_1
INA3221.h, 315
INA3221_REG_CONF_AVG_1024
INA3221.h, 315
INA3221_REG_CONF_AVG_128
INA3221.h, 315
INA3221_REG_CONF_AVG_16
INA3221.h, 315
INA3221_REG_CONF_AVG_256
INA3221.h, 315
INA3221_REG_CONF_AVG_4
INA3221.h, 315
INA3221_REG_CONF_AVG_512
INA3221.h, 315
INA3221_REG_CONF_AVG_64
INA3221.h, 315
INA3221_REG_CONF_CT_1100US
INA3221.h, 315
INA3221_REG_CONF_CT_140US
INA3221.h, 315
INA3221_REG_CONF_CT_204US
INA3221.h, 315
INA3221_REG_CONF_CT_2116US
INA3221.h, 315
INA3221_REG_CONF_CT_332US
INA3221.h, 315
INA3221_REG_CONF_CT_4156US
INA3221.h, 315
INA3221_REG_CONF_CT_588US
INA3221.h, 315
INA3221_REG_CONF_CT_8244US
INA3221.h, 315
INA3221_REG_DIE_ID
INA3221.h, 314
INA3221_REG_MANUF_ID
INA3221.h, 314
INA3221_REG_MASK_ENABLE
INA3221.h, 314
INA3221_REG_PWR_VALID_HI_LIM
INA3221.h, 314
INA3221_REG_PWR_VALID_LO_LIM
INA3221.h, 314
INA3221_REG_SHUNTV_SUM
INA3221.h, 314
INA3221_REG_SHUNTV_SUM_LIM
INA3221.h, 314
ina3221_reg_t
INA3221.h, 314
includes.h, 228
INFO
utils.h, 363
init
BH1750Wrapper, 150
BME280, 155
BME280Wrapper, 167
Event Management, 96
ISensor, 194
init_modules
main.cpp, 367
init_pico_hw
main.cpp, 366
initialize
Power Management, 116
initialize_radio
communication.cpp, 269
communication.h, 273
initialized_
BH1750Wrapper, 152
BME280, 157
BME280Wrapper, 169
PowerManager, 208
Interface
Protocol, 63
INTERNAL_FAIL_TO_READ
Protocol, 62
interval
communication.cpp, 271
INVALID_FORMAT
Protocol, 62
INVALID_OPERATION
Protocol, 61, 63
INVALID_PARAM
Protocol, 63
INVALID_VALUE
Protocol, 62
is_bootloader_reset_pending
SystemStateManager, 217
is_charging_solar
Power Management, 120
is_charging_usb
Power Management, 121
is_gps_collection_paused
SystemStateManager, 218
is_initialized
BH1750Wrapper, 150

BME280Wrapper, 167
ISensor, 194
is_sd_card_mounted
SystemStateManager, 218
is_sync_needed
RTC clock, 85
is_telemetry_collection_time
Telemetry Manager, 137
is_telemetry_flush_time
Telemetry Manager, 137
ISensor, 193
~ISensor, 194
configure, 195
get_address, 195
get_type, 194
init, 194
is_initialized, 194
read_data, 194

last_frame_sent
test_comand_handlers.cpp, 380
LAST_SYNC_TIME
clock_commands.cpp, 240
last_sync_time_
DS3231, 175
lastPrintTime
communication.cpp, 271
lastReceiveTime
communication.cpp, 271
lastSendTime
communication.cpp, 271
lat_dir
TelemetryRecord, 224
latitude
TelemetryRecord, 224
lib/clock/DS3231.cpp, 229
lib/clock/DS3231.h, 234, 238
lib/comms/commands/clock_commands.cpp, 239, 241
lib/comms/commands/commands.cpp, 243, 244
lib/comms/commands/commands.h, 245, 247
lib/comms/commands/diagnostic_commands.cpp, 247, 248
lib/comms/commands/event_commands.cpp, 250
lib/comms/commands/gps_commands.cpp, 251, 253
lib/comms/commands/power_commands.cpp, 255, 257
lib/comms/commands/sensor_commands.cpp, 259, 260
lib/comms/commands/storage_commands.cpp, 263, 265
lib/comms/commands/telemetry_commands.cpp, 266, 268
lib/comms/communication.cpp, 269, 272
lib/comms/communication.h, 272, 276
lib/comms/frame.cpp, 277, 278
lib/comms/protocol.h, 280, 282
lib/comms/receive.cpp, 283, 284
lib/comms/send.cpp, 285, 287
lib/comms/utils_converters.cpp, 287, 288
lib/eventman/event_manager.cpp, 289, 290
lib/eventman/event_manager.h, 291, 294
lib/location/gps_collector.cpp, 295, 296
lib/location/gps_collector.h, 297, 298
lib/location/NMEA/NMEA_data.h, 298, 300
lib/pin_config.h, 301, 307
lib/powerman/INA3221/INA3221.cpp, 308, 309
lib/powerman/INA3221/INA3221.h, 312, 316
lib/powerman/PowerManager.cpp, 318, 319
lib/powerman/PowerManager.h, 320, 321
lib/sensors/BH1750/BH1750.cpp, 322, 323
lib/sensors/BH1750/BH1750.h, 324, 325
lib/sensors/BH1750/BH1750_WRAPPER.cpp, 326
lib/sensors/BH1750/BH1750_WRAPPER.h, 327, 328
lib/sensors/BME280/BME280.cpp, 328, 329
lib/sensors/BME280/BME280.h, 332, 333
lib/sensors/BME280/BME280_WRAPPER.cpp, 335
lib/sensors/BME280/BME280_WRAPPER.h, 336
lib/sensors/ISensor.cpp, 337, 338
lib/sensors/ISensor.h, 338, 340
lib/storage/storage.cpp, 341
lib/storage/storage.h, 342, 343
lib/system_state_manager.h, 344, 345
lib/telemetry/telemetry_manager.cpp, 346, 348
lib/telemetry/telemetry_manager.h, 352, 354
lib/utils.cpp, 355, 360
lib/utils.h, 361, 364
LIGHT
Sensors, 123
light
SensorDataRecord, 210
LIGHT_LEVEL
Sensors, 123
LIST_FILES_COMMAND
storage_commands.cpp, 264
load_from_storage
EventManager, 183
Location, 99
collect_gps_data, 100
MAX_RAW_DATA_LENGTH, 100
splitString, 100
LOCK
Event Management, 96
log_event
Event Management, 97
LOG_FILENAME
main.cpp, 365
lon_dir
TelemetryRecord, 224
longitude
TelemetryRecord, 224
LORA
Protocol, 63
lora_address_local
pin_config.h, 307
lora_address_remote
pin_config.h, 307
lora_cs_pin
pin_config.h, 306
LORA_DEFAULT_DIO0_PIN

pin_config.h, 306
LORA_DEFAULT_RESET_PIN
pin_config.h, 306
LORA_DEFAULT_SPI
pin_config.h, 305
LORA_DEFAULT_SPI_FREQUENCY
pin_config.h, 305
LORA_DEFAULT_SS_PIN
pin_config.h, 305
lora_irq_pin
pin_config.h, 306
lora_reset_pin
pin_config.h, 306
lora_send_called
test_comand_handlers.cpp, 380
lora_tx_done_callback
communication.cpp, 270
communication.h, 274
LOST
Event Management, 96

main
main.cpp, 367
test_runner.cpp, 419
main.cpp, 365
buffer, 368
buffer_index, 368
core1_entry, 366
init_modules, 367
init_pico_hw, 366
LOG_FILENAME, 365
main, 367
MAIN_I2C_PORT
pin_config.h, 303
MAIN_I2C_SCL_PIN
pin_config.h, 303
MAIN_I2C_SDA_PIN
pin_config.h, 303
MAX_PACKET_SIZE
receive.cpp, 284
MAX_RAW_DATA_LENGTH
Location, 100
Measurement Functions, 112
get_current_ma, 113
get_shunt_voltage, 112
get_voltage, 113
MILIAMP
Protocol, 63
minutes
ds3231_data_t, 176
mock_spi_enabled
hardware_mocks.cpp, 397
hardware_mocks.h, 400
mock_spi_read_blocking
hardware_mocks.cpp, 397
hardware_mocks.h, 400
mock_spi_write_blocking
hardware_mocks.cpp, 397
hardware_mocks.h, 400
mock_uart_enabled
hardware_mocks.cpp, 397
hardware_mocks.h, 400
test_mocks.cpp, 404
mock_uart_init
hardware_mocks.cpp, 396
hardware_mocks.h, 400
mock_uart_puts
hardware_mocks.cpp, 396
hardware_mocks.h, 400
Mode
BH1750, 145
mode_bus_en
INA3221::conf_reg_t, 170
mode_continious_en
INA3221::conf_reg_t, 170
mode_shunt_en
INA3221::conf_reg_t, 170
MONDAY
DS3231.h, 237
month
ds3231_data_t, 177
MOUNT_COMMAND
storage_commands.cpp, 264
MSG_RECEIVED
Event Management, 95
MSG_SENT
Event Management, 95
msgCount
communication.cpp, 271
mutex_
SystemStateManager, 221

nextEventId
EventManager, 184
nmea_data
test_mocks.cpp, 404
NMEAData, 198
get_gga_tokens, 202
get_instance, 200
get_rmc_tokens, 202
get_unix_time, 203
gga_mutex_, 204
gga_tokens_, 203
has_valid_time, 202
NMEAData, 199
operator=, 200
rmc_mutex_, 204
rmc_tokens_, 203
update_gga_tokens, 201
update_rmc_tokens, 201
NONE
Protocol, 62, 63
Sensors, 123
NOT_ALLOWED
Protocol, 62, 63
NUM_CALIB_PARAMS
BME280, 162

on_receive
Receiving Data, 64
ONE_TIME_HIGH_RES_MODE
BH1750, 146
ONE_TIME_HIGH_RES_MODE_2
BH1750, 146
ONE_TIME_LOW_RES_MODE
BH1750, 146
operation_type_to_string
Utility Converters, 67
OperationType
Protocol, 62
operationType
Frame, 186
operator=
DS3231, 174
EventManager, 182
NMEAData, 200
PowerManager, 207
SystemStateManager, 216
outgoing
communication.cpp, 271

PA_OUTPUT_PA_BOOST_PIN
pin_config.h, 306
PA_OUTPUT_RFO_PIN
pin_config.h, 306
PARAM_INVALID
Protocol, 61
PARAM_REQUIRED
Protocol, 61
PARAM_UNECESSARY
Protocol, 63
PARAM_UNNECESSARY
Protocol, 61
PASS_THROUGH_END
Event Management, 96
PASS_THROUGH_START
Event Management, 96
PASSTHROUGH_COMMAND
gps_commands.cpp, 252
pending_bootloader_reset
SystemStateManager, 221
pin_config.h
BUFFER_SIZE, 304
DEBUG_UART_BAUD_RATE, 302
DEBUG_UART_PORT, 302
DEBUG_UART_RX_PIN, 302
DEBUG_UART_TX_PIN, 302
GPS_POWER_ENABLE_PIN, 303
GPS_UART_BAUD_RATE, 303
GPS_UART_PORT, 303
GPS_UART_RX_PIN, 303
GPS_UART_TX_PIN, 303
lora_address_local, 307
lora_address_remote, 307
lora_cs_pin, 306
LORA_DEFAULT_DIO0_PIN, 306
LORA_DEFAULT_RESET_PIN, 306
LORA_DEFAULT_SPI, 305
LORA_DEFAULT_SPI_FREQUENCY, 305
LORA_DEFAULT_SS_PIN, 305
lora_irq_pin, 306
lora_reset_pin, 306
MAIN_I2C_PORT, 303
MAIN_I2C_SCL_PIN, 303
MAIN_I2C_SDA_PIN, 303
PA_OUTPUT_PA_BOOST_PIN, 306
PA_OUTPUT_RFO_PIN, 306
READ_BIT, 305
SD_CARD_DETECT_PIN, 304
SD_CS_PIN, 304
SD_MISO_PIN, 304
SD_MOSI_PIN, 304
SD_SCK_PIN, 304
SD_SPI_PORT, 304
SPI_PORT, 305
SX1278_CS, 305
SX1278_MISO, 304
SX1278_MOSI, 305
SX1278_SCK, 305
POWER
Event Management, 94
Power Commands, 41
handle_get_current_charge_solar, 45
handle_get_current_charge_total, 46
handle_get_current_charge_usb, 44
handle_get_current_draw, 47
handle_get_power_manager_ids, 41
handle_get_voltage_5v, 43
handle_get_voltage_battery, 42
Power Management, 114
configure, 120
get_current_charge_solar, 119
get_current_charge_total, 119
get_current_charge_usb, 118
get_current_draw, 118
get_instance, 115
get_voltage_5v, 118
get_voltage_battery, 117
initialize, 116
is_charging_solar, 120
is_charging_usb, 121
PowerManager, 115
read_device_ids, 117
power_commands.cpp
CHARGE_SOLAR, 257
CHARGE_TOTAL, 257
CHARGE_USB, 256
DRAW_TOTAL, 257
POWER_GROUP, 256
POWER_MANAGER_IDS, 256
VOLTAGE_BATTERY, 256
VOLTAGE_MAIN, 256
POWER_FALLING
Event Management, 95
POWER_GROUP

power_commands.cpp, 256
POWER_MANAGER_IDS
power_commands.cpp, 256
POWER_OFF
Event Management, 96
POWER_ON
BH1750, 146
Event Management, 96
POWER_STATUS_COMMAND
gps_commands.cpp, 252
PowerEvent
Event Management, 95
powerman_mutex_
PowerManager, 208
PowerManager, 204
BATTERY_FULL_THRESHOLD, 208
BATTERY_LOW_THRESHOLD, 208
charging_solar_active_, 209
charging_usb_active_, 209
ina3221_, 208
initialized_, 208
operator=, 207
Power Management, 115
powerman_mutex_, 208
PowerManager, 206, 207
SOLAR_CURRENT_THRESHOLD, 208
USB_CURRENT_THRESHOLD, 208
powerManager
test_mocks.cpp, 403
PRESSURE
Sensors, 123
pressure
SensorDataRecord, 210
Protocol, 60
BOOL, 63
CommandAccessLevel, 62
DATETIME, 63
ERR, 62
ErrorCode, 61
ExceptionType, 63
FAIL_TO_SET, 62
GET, 62
Interface, 63
INTERNAL_FAIL_TO_READ, 62
INVALID_FORMAT, 62
INVALID_OPERATION, 61, 63
INVALID_PARAM, 63
INVALID_VALUE, 62
LORA, 63
MILIAMP, 63
NONE, 62, 63
NOT_ALLOWED, 62, 63
OperationType, 62
PARAM_INVALID, 61
PARAM_REQUIRED, 61
PARAM_UNECESSARY, 63
PARAM_UNNECESSARY, 61
READ_ONLY, 62
READ_WRITE, 62
RES, 62
SECOND, 63
SEQ, 62
SET, 62
TEXT, 63
UART, 63
UNDEFINED, 63
UNKNOWN_ERROR, 62
VAL, 62
ValueUnit, 62
VOLT, 63
WRITE_ONLY, 62
protocol.h
DELIMITER, 281
FRAME_BEGIN, 281
FRAME_END, 281
pwr_valid_alert
INA3221::masken_reg_t, 196

RADIO_ERROR
Event Management, 95
RADIO_INIT
Event Management, 95
READ_BIT
pin_config.h, 305
read_data
BH1750Wrapper, 150
BME280Wrapper, 167
ISensor, 194
read_device_ids
Power Management, 117
READ_ONLY
Protocol, 62
read_raw_all
BME280, 155
read_register
Configuration Functions, 106
read_temperature
RTC clock, 75
READ_WRITE
Protocol, 62
receive.cpp
MAX_PACKET_SIZE, 284
Receiving Data, 63
handle_uart_input, 64
on_receive, 64
refactor, 1
refactor.md, 371
REG_CONFIG
BME280, 157
REG_CTRL_HUM
BME280, 157
REG_CTRL_MEAS
BME280, 157
REG_DIG_H1
BME280, 161
REG_DIG_H2
BME280, 161

REG_DIG_H3
BME280, 161
REG_DIG_H4
BME280, 161
REG_DIG_H5
BME280, 162
REG_DIG_H6
BME280, 162
REG_DIG_P1_LSB
BME280, 159
REG_DIG_P1_MSB
BME280, 159
REG_DIG_P2_LSB
BME280, 159
REG_DIG_P2_MSB
BME280, 159
REG_DIG_P3_LSB
BME280, 159
REG_DIG_P3_MSB
BME280, 159
REG_DIG_P4_LSB
BME280, 160
REG_DIG_P4_MSB
BME280, 160
REG_DIG_P5_LSB
BME280, 160
REG_DIG_P5_MSB
BME280, 160
REG_DIG_P6_LSB
BME280, 160
REG_DIG_P6_MSB
BME280, 160
REG_DIG_P7_LSB
BME280, 160
REG_DIG_P7_MSB
BME280, 160
REG_DIG_P8_LSB
BME280, 161
REG_DIG_P8_MSB
BME280, 161
REG_DIG_P9_LSB
BME280, 161
REG_DIG_P9_MSB
BME280, 161
REG_DIG_T1_LSB
BME280, 158
REG_DIG_T1_MSB
BME280, 158
REG_DIG_T2_LSB
BME280, 158
REG_DIG_T2_MSB
BME280, 158
REG_DIG_T3_LSB
BME280, 159
REG_DIG_T3_MSB
BME280, 159
REG_HUMIDITY_MSB
BME280, 158
REG_PRESSURE_MSB
BME280, 158
REG_RESET
BME280, 158
REG_TEMPERATURE_MSB
BME280, 158
RES
Protocol, 62
reserved
INA3221::masken_reg_t, 198
RESET
BH1750, 146
reset
BME280, 155
Configuration Functions, 104
INA3221::conf_reg_t, 171
RMC_DATA_COMMAND
gps_commands.cpp, 252
rmc_mutex_
NMEAData, 204
rmc_tokens_
NMEAData, 203
RTC clock, 71
bcd_to_bin, 91
bin_to_bcd, 89
clock_enable, 79
DS3231, 72
get_clock_sync_interval, 81
get_instance, 72
get_last_sync_time, 83
get_local_time, 84
get_time, 74
get_timezone_offset, 80
get_unix_time, 78
i2c_read_reg, 87
i2c_write_reg, 88
is_sync_needed, 85
read_temperature, 75
set_clock_sync_interval, 82
set_time, 73
set_timezone_offset, 80
set_unix_time, 77
sync_clock_with_gps, 86
update_last_sync_time, 83

sample_interval_ms
Telemetry Manager, 140
satellites
TelemetryRecord, 225
SATURDAY
DS3231.h, 237
save_to_storage
Event Management, 98
scan_connected_sensors
Sensors, 125
SD_CARD_DETECT_PIN
pin_config.h, 304
sd_card_mounted
SystemStateManager, 221

SD_CS_PIN
pin_config.h, 304
SD_MISO_PIN
pin_config.h, 304
SD_MOSI_PIN
pin_config.h, 304
SD_SCK_PIN
pin_config.h, 304
SD_SPI_PORT
pin_config.h, 304
SECOND
Protocol, 63
seconds
ds3231_data_t, 176
send.cpp
send_frame_lora, 286
send_frame_uart, 286
send_message, 286
send_frame_lora
communication.h, 275
send.cpp, 286
test_comand_handlers.cpp, 377
send_frame_uart
communication.h, 274
send.cpp, 286
test_comand_handlers.cpp, 377
send_message
communication.h, 274
send.cpp, 286
Sensor Commands, 48
handle_get_sensor_data, 48
handle_get_sensor_list, 50
handle_sensor_config, 49
sensor_
BH1750Wrapper, 152
BME280Wrapper, 169
sensor_commands.cpp
SENSOR_CONFIGURE, 260
SENSOR_GROUP, 260
SENSOR_READ, 260
SENSOR_CONFIGURE
sensor_commands.cpp, 260
sensor_configure
Sensors, 123
sensor_data_buffer
Telemetry Manager, 141, 142
sensor_data_buffer_count
Telemetry Manager, 141, 142
sensor_data_buffer_write_index
Telemetry Manager, 141, 142
SENSOR_DATA_CSV_PATH
Telemetry Manager, 130
SENSOR_GROUP
sensor_commands.cpp, 260
sensor_init
Sensors, 123
SENSOR_READ
sensor_commands.cpp, 260
sensor_read_data
Sensors, 124
SensorDataRecord, 209
humidity, 210
light, 210
pressure, 210
temperature, 210
timestamp, 210
SensorDataTypeIdentifier
Sensors, 123
Sensors, 121
ENVIRONMENT, 123
get_available_sensors, 125
get_sensor, 124
HUMIDITY, 123
LIGHT, 123
LIGHT_LEVEL, 123
NONE, 123
PRESSURE, 123
scan_connected_sensors, 125
sensor_configure, 123
sensor_init, 123
sensor_read_data, 124
SensorDataTypeIdentifier, 123
SensorType, 122
TEMPERATURE, 123
sensors
SensorWrapper, 213
SensorType
Sensors, 122
SensorWrapper, 211
get_instance, 212
sensors, 213
SensorWrapper, 212
SEQ
Protocol, 62
SET
Protocol, 62
set_averaging_mode
Configuration Functions, 110
set_bootloader_reset_pending
SystemStateManager, 217
set_bus_conversion_time
Configuration Functions, 110
set_bus_measurement_disable
Configuration Functions, 109
set_bus_measurement_enable
Configuration Functions, 109
set_clock_sync_interval
RTC clock, 82
set_gps_collection_paused
SystemStateManager, 218
set_mode_continuous
Configuration Functions, 107
set_mode_power_down
Configuration Functions, 106
set_mode_triggered
Configuration Functions, 107

set_sd_card_mounted
SystemStateManager, 219
set_shunt_conversion_time
Configuration Functions, 111
set_shunt_measurement_disable
Configuration Functions, 108
set_shunt_measurement_enable
Configuration Functions, 108
set_time
RTC clock, 73
set_timezone_offset
RTC clock, 80
set_uart_verbosity
SystemStateManager, 220
set_unix_time
RTC clock, 77
setUp
test_comand_handlers.cpp, 378
test_frame_send.cpp, 394
shunt_conv_time
INA3221::conf_reg_t, 170
shunt_sum_alert
INA3221::masken_reg_t, 197
shunt_sum_en_ch1
INA3221::masken_reg_t, 198
shunt_sum_en_ch2
INA3221::masken_reg_t, 198
shunt_sum_en_ch3
INA3221::masken_reg_t, 197
SHUNT_VOLTAGE_LSB_UV
INA3221.h, 315
SHUTDOWN
Event Management, 95
SILENT
utils.h, 363
SOLAR_ACTIVE
Event Management, 95
SOLAR_CURRENT_THRESHOLD
PowerManager, 208
SOLAR_INACTIVE
Event Management, 95
speed
TelemetryRecord, 224
spi_output_buffer
hardware_mocks.cpp, 397
hardware_mocks.h, 401
SPI_PORT
pin_config.h, 305
split_and_send_message
communication.h, 276
splitString
Location, 100
Storage, 126
fs_init, 127
fs_stop, 127
Storage Commands, 51
handle_list_files, 52
handle_mount, 52
storage_commands.cpp
LIST_FILES_COMMAND, 264
MOUNT_COMMAND, 264
STORAGE_GROUP, 264
STORAGE_GROUP
storage_commands.cpp, 264
string_to_operation_type
Utility Converters, 68
SUNDAY
DS3231.h, 237
SX1278_CS
pin_config.h, 305
SX1278_MISO
pin_config.h, 304
SX1278_MOSI
pin_config.h, 305
SX1278_SCK
pin_config.h, 305
sync_clock_with_gps
RTC clock, 86
sync_interval_minutes_
DS3231, 175
SYSTEM
Event Management, 94
System State Manager, 128
system_voltage
TelemetryRecord, 223
systemClock
test_mocks.cpp, 403
SystemEvent
Event Management, 94
SystemStateManager, 213
get_instance, 216
get_uart_verbosity, 219
gps_collection_paused, 221
is_bootloader_reset_pending, 217
is_gps_collection_paused, 218
is_sd_card_mounted, 218
mutex_, 221
operator=, 216
pending_bootloader_reset, 221
sd_card_mounted, 221
set_bootloader_reset_pending, 217
set_gps_collection_paused, 218
set_sd_card_mounted, 219
set_uart_verbosity, 220
SystemStateManager, 214, 215
uart_verbosity, 221

t_fine
BME280, 157
tearDown
test_comand_handlers.cpp, 378
test_frame_send.cpp, 394
Telemetry Buffer Commands, 53
handle_get_last_sensor_record, 54
handle_get_last_telemetry_record, 54
Telemetry Manager, 128
collect_gps_telemetry, 133

collect_power_telemetry, 131
collect_sensor_telemetry, 134
collect_telemetry, 134
DEFAULT_FLUSH_THRESHOLD, 131
DEFAULT_SAMPLE_INTERVAL_MS, 130
emit_power_events, 132
flush_sensor_data, 136
flush_telemetry, 135
flush_threshold, 140
get_last_sensor_record_csv, 139
get_last_telemetry_record_csv, 138
is_telemetry_collection_time, 137
is_telemetry_flush_time, 137
sample_interval_ms, 140
sensor_data_buffer, 141, 142
sensor_data_buffer_count, 141, 142
sensor_data_buffer_write_index, 141, 142
SENSOR_DATA_CSV_PATH, 130
telemetry_buffer, 141, 142
telemetry_buffer_count, 141, 142
TELEMETRY_BUFFER_SIZE, 142, 143
telemetry_buffer_write_index, 141, 142
TELEMETRY_CSV_PATH, 130
telemetry_init, 131
telemetry_mutex, 141, 143
to_csv, 139, 140
telemetry_buffer
Telemetry Manager, 141, 142
telemetry_buffer_count
Telemetry Manager, 141, 142
TELEMETRY_BUFFER_SIZE
Telemetry Manager, 142, 143
telemetry_buffer_write_index
Telemetry Manager, 141, 142
telemetry_commands.cpp
GET_LAST_SENSOR_RECORD_COMMAND, 267
GET_LAST_TELEMETRY_RECORD_COMMAND, 267
TELEMETRY_GROUP, 267
TELEMETRY_CSV_PATH
Telemetry Manager, 130
TELEMETRY_GROUP
telemetry_commands.cpp, 267
telemetry_init
Telemetry Manager, 131
telemetry_mutex
Telemetry Manager, 141, 143
TelemetryRecord, 222
altitude, 225
battery_voltage, 223
build_version, 222
charge_current_solar, 223
charge_current_usb, 223
course, 224
date, 225
discharge_current, 223
fix_quality, 225
lat_dir, 224
latitude, 224
lon_dir, 224
longitude, 224
satellites, 225
speed, 224
system_voltage, 223
time, 223
timestamp, 222
TEMPERATURE
Sensors, 123
temperature
SensorDataRecord, 210
test/comms/commands/test_clock_commands.cpp, 371
test/comms/commands/test_diagnostic_commands.cpp, 371, 374
test/comms/commands/test_event_commands.cpp, 375
test/comms/commands/test_gps_commands.cpp, 375
test/comms/commands/test_power_commands.cpp, 375
test/comms/commands/test_sensor_commands.cpp, 375
test/comms/commands/test_storage_commands.cpp, 376
test/comms/commands/test_telemetry_commands.cpp, 376
test/comms/test_comand_handlers.cpp, 376, 381
test/comms/test_converters.cpp, 382, 384
test/comms/test_frame_build.cpp, 385, 389
test/comms/test_frame_coding.cpp, 390, 392
test/comms/test_frame_common.h, 392, 394
test/comms/test_frame_send.cpp, 394, 395
test/mocks/hardware_mocks.cpp, 396, 398
test/mocks/hardware_mocks.h, 398, 401
test/test_mocks.cpp, 401, 405
test/test_runner.cpp, 406, 420
test_comand_handlers.cpp
last_frame_sent, 380
lora_send_called, 380
send_frame_lora, 377
send_frame_uart, 377
setUp, 378
tearDown, 378
test_command_handler_get_operation, 378
test_command_handler_invalid_operation, 379
test_command_handler_set_operation, 379
uart_send_called, 380
test_command_handler_get_operation
test_comand_handlers.cpp, 378
test_mocks.cpp, 403
test_runner.cpp, 414
test_command_handler_invalid_operation
test_comand_handlers.cpp, 379
test_mocks.cpp, 403
test_runner.cpp, 415
test_command_handler_set_operation
test_comand_handlers.cpp, 379
test_mocks.cpp, 403

test_runner.cpp, 415
test_converters.cpp
test_exception_type_conversion, 383
test_hex_string_conversion, 384
test_operation_type_conversion, 382
test_value_unit_type_conversion, 382
test_diagnostic_commands.cpp
test_handle_enter_bootloader_mode, 373
test_handle_get_build_version, 372
test_handle_get_commands_list, 372
test_handle_verbosity, 373
test_error_code_conversion
test_mocks.cpp, 402
test_runner.cpp, 419
test_exception_type_conversion
test_converters.cpp, 383
test_runner.cpp, 413
test_frame_build.cpp
test_frame_build_err, 386
test_frame_build_get, 386
test_frame_build_res, 387
test_frame_build_seq, 388
test_frame_build_set, 387
test_frame_build_val, 385
test_frame_build_err
test_frame_build.cpp, 386
test_runner.cpp, 411
test_frame_build_get
test_frame_build.cpp, 386
test_runner.cpp, 408
test_frame_build_res
test_frame_build.cpp, 387
test_runner.cpp, 409
test_frame_build_seq
test_frame_build.cpp, 388
test_runner.cpp, 410
test_frame_build_set
test_frame_build.cpp, 387
test_runner.cpp, 409
test_frame_build_val
test_frame_build.cpp, 385
test_runner.cpp, 411
test_frame_coding.cpp
test_frame_decode_basic, 390
test_frame_decode_invalid_header, 391
test_frame_encode_basic, 390
test_frame_common.h
create_test_frame, 393
test_frame_decode_basic
test_frame_coding.cpp, 390
test_runner.cpp, 407
test_frame_decode_invalid_header
test_frame_coding.cpp, 391
test_runner.cpp, 408
test_frame_encode_basic
test_frame_coding.cpp, 390
test_runner.cpp, 407
test_frame_send.cpp
setUp, 394
tearDown, 394
test_send_frame_uart, 395
test_handle_enter_bootloader_mode
test_diagnostic_commands.cpp, 373
test_runner.cpp, 418
test_handle_get_build_version
test_diagnostic_commands.cpp, 372
test_runner.cpp, 417
test_handle_get_commands_list
test_diagnostic_commands.cpp, 372
test_runner.cpp, 416
test_handle_verbosity
test_diagnostic_commands.cpp, 373
test_runner.cpp, 417
test_hex_string_conversion
test_converters.cpp, 384
test_runner.cpp, 413
test_mocks.cpp
g_pending_bootloader_reset, 404
g_uart_verbosity, 404
get_last_sensor_record_csv, 402
get_last_telemetry_record_csv, 402
mock_uart_enabled, 404
nmea_data, 404
powerManager, 403
systemClock, 403
test_command_handler_get_operation, 403
test_command_handler_invalid_operation, 403
test_command_handler_set_operation, 403
test_error_code_conversion, 402
uart_output_buffer, 404
uart_print, 402
test_operation_type_conversion
test_converters.cpp, 382
test_runner.cpp, 411
test_runner.cpp
main, 419
test_command_handler_get_operation, 414
test_command_handler_invalid_operation, 415
test_command_handler_set_operation, 415
test_error_code_conversion, 419
test_exception_type_conversion, 413
test_frame_build_err, 411
test_frame_build_get, 408
test_frame_build_res, 409
test_frame_build_seq, 410
test_frame_build_set, 409
test_frame_build_val, 411
test_frame_decode_basic, 407
test_frame_decode_invalid_header, 408
test_frame_encode_basic, 407
test_handle_enter_bootloader_mode, 418
test_handle_get_build_version, 417
test_handle_get_commands_list, 416
test_handle_verbosity, 417
test_hex_string_conversion, 413
test_operation_type_conversion, 411

test_value_unit_type_conversion, 412
test_send_frame_uart
test_frame_send.cpp, 395
test_value_unit_type_conversion
test_converters.cpp, 382
test_runner.cpp, 412
TEXT
Protocol, 63
THURSDAY
DS3231.h, 237
TIME
clock_commands.cpp, 240
time
TelemetryRecord, 223
timestamp
event_manager.h, 293
EventLog, 180
SensorDataRecord, 210
TelemetryRecord, 222
TIMEZONE_OFFSET
clock_commands.cpp, 240
timezone_offset_minutes_
DS3231, 175
timing_ctrl_alert
INA3221::masken_reg_t, 196
to_csv
Telemetry Manager, 139, 140
todo, 9
todo.md, 421
TUESDAY
DS3231.h, 237

UART
Protocol, 63
UART_ERROR
Event Management, 95
uart_mutex
utils.cpp, 360
uart_output_buffer
hardware_mocks.cpp, 397
hardware_mocks.h, 400
test_mocks.cpp, 404
uart_print
test_mocks.cpp, 402
utils.cpp, 358
utils.h, 363
uart_send_called
test_comand_handlers.cpp, 380
uart_verbosity
SystemStateManager, 221
UNCONFIGURED_POWER_DOWN
BH1750, 146
UNDEFINED
Protocol, 63
unit
Frame, 186
UNKNOWN_ERROR
Protocol, 62
update_gga_tokens
NMEAData, 201
update_last_sync_time
RTC clock, 83
update_rmc_tokens
NMEAData, 201
USB_CONNECTED
Event Management, 95
USB_CURRENT_THRESHOLD
PowerManager, 208
USB_DISCONNECTED
Event Management, 95
Utility Converters, 65
error_code_to_string, 68
exception_type_to_string, 66
hex_string_to_bytes, 70
operation_type_to_string, 67
string_to_operation_type, 68
value_unit_type_to_string, 66
utils.cpp
get_level_color, 356
get_level_prefix, 357
uart_mutex, 360
uart_print, 358
utils.h
ANSI_BLUE, 362
ANSI_GREEN, 362
ANSI_RED, 362
ANSI_RESET, 362
ANSI_YELLOW, 362
DEBUG, 363
ERROR, 363
INFO, 363
SILENT, 363
uart_print, 363
VerbosityLevel, 362
WARNING, 363

VAL
Protocol, 62
value
Frame, 186
value_unit_type_to_string
Utility Converters, 66
ValueUnit
Protocol, 62
VerbosityLevel
utils.h, 362
VOLT
Protocol, 63
VOLTAGE_BATTERY
power_commands.cpp, 256
VOLTAGE_MAIN
power_commands.cpp, 256

warn_alert_ch1
INA3221::masken_reg_t, 197
warn_alert_ch2
INA3221::masken_reg_t, 196
warn_alert_ch3

INA3221::masken_reg_t, 196
warn_alert_latch_en
INA3221::masken_reg_t, 197
WARNING
utils.h, 363
WATCHDOG_RESET
Event Management, 95
WEDNESDAY
DS3231.h, 237
write8
BH1750, 147
WRITE_ONLY
Protocol, 62
writeIndex
EventManager, 184

year
ds3231_data_t, 177