

KubiSat Firmware

Generated by Doxygen 1.13.2

1 credits	1
1.1 pico-vfs	1
1.2 pico-lora	2
2 Clock Commands	3
3 Topic Index	5
3.1 Topics	5
4 Hierarchical Index	7
4.1 Class Hierarchy	7
5 Class Index	9
5.1 Class List	9
6 File Index	11
6.1 File List	11
7 Topic Documentation	13
7.1 Clock Management Commands	13
7.1.1 Detailed Description	13
7.1.2 Function Documentation	13
7.1.2.1 handle_time()	13
7.1.2.2 handle_timezone_offset()	14
7.1.2.3 handle_get_internal_temperature()	15
7.2 Command System	16
7.2.1 Detailed Description	16
7.2.2 Macro Definition Documentation	17
7.2.2.1 CMD	17
7.2.3 Typedef Documentation	17
7.2.3.1 CommandHandler	17
7.2.3.2 CommandMap	17
7.2.4 Function Documentation	17
7.2.4.1 execute_command()	17
7.2.5 Variable Documentation	18
7.2.5.1 command_handlers	18
7.3 Diagnostic Commands	19
7.3.1 Detailed Description	19
7.3.2 Function Documentation	19
7.3.2.1 handle_get_commands_list()	19
7.3.2.2 handle_get_build_version()	20
7.3.2.3 handle_get_uptime()	21
7.3.2.4 handle_get_power_mode()	21
7.3.2.5 handle_verbosity()	22

7.3.2.6 handle_enter_bootloader_mode()	23
7.3.3 Variable Documentation	24
7.3.3.1 diagnostic_commands_group_id	24
7.3.3.2 commands_list_command_id	24
7.3.3.3 build_version_command_id	24
7.3.3.4 power_mode_command_id	25
7.3.3.5 uptime_command_id	25
7.3.3.6 verbosity_command_id	25
7.3.3.7 enter_bootloader_command_id	25
7.4 Event Commands	25
7.4.1 Detailed Description	25
7.4.2 Function Documentation	25
7.4.2.1 handle_get_last_events()	25
7.4.2.2 handle_get_event_count()	26
7.4.3 Variable Documentation	27
7.4.3.1 event_commands_group_id	27
7.4.3.2 last_events_command_id	27
7.4.3.3 event_count_command_id	28
7.5 GPS Commands	28
7.5.1 Detailed Description	28
7.5.2 Function Documentation	28
7.5.2.1 handle_gps_power_status()	28
7.5.2.2 handle_enable_gps_uart_passthrough()	29
7.6 Telemetry Buffer Commands	30
7.6.1 Detailed Description	30
7.6.2 Function Documentation	30
7.6.2.1 handle_get_last_telemetry_record()	30
7.6.2.2 handle_get_last_sensor_record()	31
7.7 Frame Handling	32
7.7.1 Detailed Description	32
7.7.2 Function Documentation	32
7.7.2.1 frame_encode()	32
7.7.2.2 frame_decode()	33
7.7.2.3 frame_process()	34
7.7.2.4 frame_build()	35
7.8 Protocol	36
7.8.1 Detailed Description	37
7.8.2 Enumeration Type Documentation	37
7.8.2.1 ErrorCode	37
7.8.2.2 OperationType	38
7.8.2.3 ValueUnit	38
7.8.2.4 Interface	38

7.9 Utility Converters	39
7.9.1 Detailed Description	39
7.9.2 Function Documentation	39
7.9.2.1 value_unit_type_to_string()	39
7.9.2.2 operation_type_to_string()	40
7.9.2.3 string_to_operation_type()	41
7.9.2.4 error_code_to_string()	41
7.10 RTC clock	42
7.10.1 Detailed Description	43
7.10.2 Function Documentation	43
7.10.2.1 DS3231()	43
7.10.2.2 get_instance()	44
7.10.2.3 get_time()	45
7.10.2.4 set_time()	45
7.10.2.5 read_temperature()	46
7.10.2.6 get_timezone_offset()	47
7.10.2.7 set_timezone_offset()	47
7.10.2.8 get_local_time()	48
7.10.2.9 i2c_read_reg()	49
7.10.2.10 i2c_write_reg()	50
7.11 Event Management	51
7.11.1 Detailed Description	53
7.11.2 Macro Definition Documentation	53
7.11.2.1 EVENT_BUFFER_SIZE	53
7.11.2.2 EVENT_FLUSH_THRESHOLD	53
7.11.2.3 EVENT_LOG_FILE	53
7.11.3 Enumeration Type Documentation	53
7.11.3.1 EventGroup	53
7.11.3.2 SystemEvent	54
7.11.3.3 PowerEvent	54
7.11.3.4 CommsEvent	55
7.11.3.5 GPSEvent	55
7.11.3.6 ClockEvent	55
7.11.4 Function Documentation	56
7.11.4.1 __attribute__().	56
7.11.4.2 init()	56
7.11.4.3 log_event()	56
7.11.4.4 get_event()	57
7.11.4.5 save_to_storage()	58
7.11.5 Variable Documentation	58
7.11.5.1 __attribute__().	58
7.12 Location	58

7.12.1 Detailed Description	59
7.12.2 Macro Definition Documentation	59
7.12.2.1 MAX_RAW_DATA_LENGTH	59
7.12.3 Function Documentation	59
7.12.3.1 splitString()	59
7.12.3.2 collect_gps_data()	60
7.13 INA3221 Power Monitor	61
7.13.1 Detailed Description	61
7.13.2 Configuration Functions	61
7.13.2.1 Detailed Description	62
7.13.2.2 Function Documentation	62
7.13.3 Measurement Functions	66
7.13.3.1 Detailed Description	66
7.13.3.2 Function Documentation	66
7.14 Power Management	68
7.14.1 Detailed Description	69
7.14.2 Function Documentation	69
7.14.2.1 PowerManager()	69
7.14.2.2 get_instance()	70
7.14.2.3 initialize()	70
7.14.2.4 read_device_ids()	71
7.14.2.5 get_voltage_battery()	71
7.14.2.6 get_voltage_5v()	71
7.14.2.7 get_voltage_solar()	72
7.14.2.8 get_current_charge_usb()	72
7.14.2.9 get_current_draw()	72
7.14.2.10 get_current_charge_solar()	73
7.14.2.11 get_current_charge_total()	73
7.14.2.12 configure()	73
7.15 BH1750 Light Sensor	74
7.15.1 Detailed Description	74
7.15.2 Function Documentation	75
7.15.2.1 BH1750()	75
7.15.2.2 begin()	75
7.15.2.3 configure()	75
7.15.2.4 get_light_level()	76
7.15.2.5 write8()	76
7.15.3 Constants	77
7.15.3.1 Detailed Description	77
7.15.3.2 Macro Definition Documentation	78
7.15.4 Types	78
7.15.4.1 Detailed Description	79

7.15.4.2 Enumeration Type Documentation	79
7.16 Sensors	79
7.16.1 Detailed Description	80
7.16.2 Enumeration Type Documentation	80
7.16.2.1 SensorType	80
7.16.2.2 SensorDataTypelIdentifier	80
7.16.3 Function Documentation	81
7.16.3.1 sensor_init()	81
7.16.3.2 sensor_configure()	81
7.16.3.3 sensor_read_data()	82
7.16.3.4 get_sensor()	82
7.17 Storage	83
7.17.1 Detailed Description	83
7.17.2 Function Documentation	83
7.17.2.1 fs_init()	83
7.17.2.2 fs_stop()	84
7.18 System State Manager	84
7.18.1 Detailed Description	85
7.18.2 Enumeration Type Documentation	85
7.18.2.1 SystemOperatingMode	85
7.19 Telemetry Manager	85
7.19.1 Detailed Description	86
7.19.2 Macro Definition Documentation	86
7.19.2.1 TELEMETRY_CSV_PATH	86
7.19.2.2 SENSOR_DATA_CSV_PATH	86
7.19.2.3 DEFAULT_SAMPLE_INTERVAL_MS	87
7.19.2.4 DEFAULT_FLUSH_THRESHOLD	87
7.19.3 Function Documentation	87
7.19.3.1 to_csv() [1/2]	87
7.19.3.2 to_csv() [2/2]	87
7.19.3.3 collect_power_telemetry()	87
7.19.3.4 emit_power_events()	88
7.19.3.5 collect_gps_telemetry()	89
7.19.3.6 collect_sensor_telemetry()	90
7.19.3.7 TelemetryManager()	90
7.19.3.8 init()	91
7.19.3.9 collect_telemetry()	92
7.19.3.10 flush_telemetry()	92
7.19.3.11 is_telemetry_collection_time()	93
7.19.3.12 is_telemetry_flush_time()	94
7.19.3.13 get_last_telemetry_record_csv()	94
7.19.3.14 get_last_sensor_record_csv()	95

8 Class Documentation	97
8.1 BH1750 Class Reference	97
8.1.1 Detailed Description	98
8.1.2 Member Data Documentation	98
8.1.2.1 _i2c_addr	98
8.1.2.2 i2c_port_	98
8.2 BH1750Wrapper Class Reference	99
8.2.1 Detailed Description	100
8.2.2 Constructor & Destructor Documentation	100
8.2.2.1 BH1750Wrapper() [1/2]	100
8.2.2.2 BH1750Wrapper() [2/2]	100
8.2.3 Member Function Documentation	100
8.2.3.1 get_i2c_addr()	100
8.2.3.2 init()	100
8.2.3.3 read_data()	100
8.2.3.4 is_initialized()	101
8.2.3.5 get_type()	101
8.2.3.6 configure()	101
8.2.3.7 get_address()	102
8.2.4 Member Data Documentation	102
8.2.4.1 sensor_	102
8.2.4.2 initialized_	102
8.3 BME280 Class Reference	103
8.3.1 Detailed Description	105
8.3.2 Member Enumeration Documentation	105
8.3.2.1 anonymous enum	105
8.3.2.2 Oversampling	105
8.3.2.3 anonymous enum	105
8.3.2.4 anonymous enum	106
8.3.2.5 anonymous enum	107
8.3.3 Constructor & Destructor Documentation	107
8.3.3.1 BME280()	107
8.3.4 Member Function Documentation	108
8.3.4.1 init()	108
8.3.4.2 reset()	108
8.3.4.3 read_raw_all()	108
8.3.4.4 convert_temperature()	109
8.3.4.5 convert_pressure()	109
8.3.4.6 convert_humidity()	110
8.3.4.7 write_register()	110
8.3.4.8 read_register() [1/2]	111
8.3.4.9 read_register() [2/2]	111

8.3.4.10 <code>configure_sensor()</code>	112
8.3.4.11 <code>get_calibration_parameters()</code>	113
8.3.5 Member Data Documentation	113
8.3.5.1 <code>i2c_port</code>	113
8.3.5.2 <code>device_addr</code>	113
8.3.5.3 <code>calib_params</code>	114
8.3.5.4 <code>initialized_</code>	114
8.3.5.5 <code>t_fine</code>	114
8.4 BME280CalibParam Struct Reference	114
8.4.1 Detailed Description	115
8.4.2 Member Data Documentation	116
8.4.2.1 <code>dig_t1</code>	116
8.4.2.2 <code>dig_t2</code>	116
8.4.2.3 <code>dig_t3</code>	116
8.4.2.4 <code>dig_p1</code>	116
8.4.2.5 <code>dig_p2</code>	116
8.4.2.6 <code>dig_p3</code>	116
8.4.2.7 <code>dig_p4</code>	117
8.4.2.8 <code>dig_p5</code>	117
8.4.2.9 <code>dig_p6</code>	117
8.4.2.10 <code>dig_p7</code>	117
8.4.2.11 <code>dig_p8</code>	117
8.4.2.12 <code>dig_p9</code>	117
8.4.2.13 <code>dig_h1</code>	118
8.4.2.14 <code>dig_h2</code>	118
8.4.2.15 <code>dig_h3</code>	118
8.4.2.16 <code>dig_h4</code>	118
8.4.2.17 <code>dig_h5</code>	118
8.4.2.18 <code>dig_h6</code>	118
8.5 BME280Wrapper Class Reference	119
8.5.1 Detailed Description	120
8.5.2 Constructor & Destructor Documentation	120
8.5.2.1 <code>BME280Wrapper()</code>	120
8.5.3 Member Function Documentation	120
8.5.3.1 <code>init()</code>	120
8.5.3.2 <code>read_data()</code>	120
8.5.3.3 <code>is_initialized()</code>	121
8.5.3.4 <code>get_type()</code>	121
8.5.3.5 <code>configure()</code>	121
8.5.3.6 <code>get_address()</code>	122
8.5.4 Member Data Documentation	122
8.5.4.1 <code>sensor_</code>	122

8.5.4.2 initialized_	122
8.6 INA3221::conf_reg_t Struct Reference	122
8.6.1 Detailed Description	123
8.6.2 Member Data Documentation	123
8.6.2.1 mode_shunt_en	123
8.6.2.2 mode_bus_en	123
8.6.2.3 mode_continious_en	123
8.6.2.4 shunt_conv_time	123
8.6.2.5 bus_conv_time	124
8.6.2.6 avg_mode	124
8.6.2.7 ch3_en	124
8.6.2.8 ch2_en	124
8.6.2.9 ch1_en	124
8.6.2.10 reset	124
8.7 DS3231 Class Reference	124
8.7.1 Detailed Description	125
8.7.2 Constructor & Destructor Documentation	126
8.7.2.1 DS3231() [1/2]	126
8.7.2.2 DS3231() [2/2]	126
8.7.3 Member Function Documentation	127
8.7.3.1 operator=()	127
8.7.4 Member Data Documentation	127
8.7.4.1 i2c	127
8.7.4.2 ds3231_addr	127
8.7.4.3 clock_mutex_	128
8.7.4.4 timezone_offset_minutes_	128
8.7.4.5 sync_interval_minutes_	128
8.7.4.6 last_sync_time_	128
8.8 ds3231_data_t Struct Reference	128
8.8.1 Detailed Description	129
8.8.2 Member Data Documentation	129
8.8.2.1 seconds	129
8.8.2.2 minutes	129
8.8.2.3 hours	129
8.8.2.4 day	129
8.8.2.5 date	129
8.8.2.6 month	130
8.8.2.7 year	130
8.8.2.8 century	130
8.9 EventEmitter Class Reference	130
8.9.1 Detailed Description	130
8.9.2 Member Function Documentation	130

8.9.2.1 emit()	130
8.10 EventLog Class Reference	131
8.10.1 Detailed Description	132
8.10.2 Member Data Documentation	132
8.10.2.1 timestamp	132
8.10.2.2 id	132
8.10.2.3 group	132
8.10.2.4 event	133
8.11 EventManager Class Reference	133
8.11.1 Detailed Description	134
8.11.2 Constructor & Destructor Documentation	134
8.11.2.1 EventManager() [1/2]	134
8.11.2.2 EventManager() [2/2]	135
8.11.3 Member Function Documentation	135
8.11.3.1 operator=()	135
8.11.3.2 get_instance()	135
8.11.3.3 get_event_count()	136
8.11.4 Member Data Documentation	136
8.11.4.1 events	136
8.11.4.2 eventCount	137
8.11.4.3 writeIndex	137
8.11.4.4 eventMutex	137
8.11.4.5 nextEventId	137
8.11.4.6 eventsSinceFlush	137
8.12 Frame Struct Reference	137
8.12.1 Detailed Description	138
8.12.2 Member Data Documentation	138
8.12.2.1 header	138
8.12.2.2 direction	138
8.12.2.3 operationType	138
8.12.2.4 group	139
8.12.2.5 command	139
8.12.2.6 value	139
8.12.2.7 unit	139
8.12.2.8 footer	139
8.13 INA3221 Class Reference	139
8.13.1 Detailed Description	141
8.13.2 Member Function Documentation	141
8.13.2.1 _read()	141
8.13.2.2 _write()	141
8.13.3 Member Data Documentation	142
8.13.3.1 _i2c_addr	142

8.13.3.2 _i2c	142
8.13.3.3 _shuntRes	142
8.13.3.4 _filterRes	142
8.13.3.5 _masken_reg	143
8.14 ISensor Class Reference	143
8.14.1 Detailed Description	144
8.14.2 Constructor & Destructor Documentation	144
8.14.2.1 ~ISensor()	144
8.14.3 Member Function Documentation	144
8.14.3.1 init()	144
8.14.3.2 read_data()	144
8.14.3.3 is_initialized()	145
8.14.3.4 get_type()	145
8.14.3.5 configure()	145
8.14.3.6 get_address()	145
8.15 INA3221::masken_reg_t Struct Reference	146
8.15.1 Detailed Description	146
8.15.2 Member Data Documentation	146
8.15.2.1 conv_ready	146
8.15.2.2 timing_ctrl_alert	146
8.15.2.3 pwr_valid_alert	146
8.15.2.4 warn_alert_ch3	147
8.15.2.5 warn_alert_ch2	147
8.15.2.6 warn_alert_ch1	147
8.15.2.7 shunt_sum_alert	147
8.15.2.8 crit_alert_ch3	147
8.15.2.9 crit_alert_ch2	147
8.15.2.10 crit_alert_ch1	147
8.15.2.11 crit_alert_latch_en	147
8.15.2.12 warn_alert_latch_en	148
8.15.2.13 shunt_sum_en_ch3	148
8.15.2.14 shunt_sum_en_ch2	148
8.15.2.15 shunt_sum_en_ch1	148
8.15.2.16 reserved	148
8.16 NMEAData Class Reference	148
8.16.1 Detailed Description	149
8.16.2 Constructor & Destructor Documentation	149
8.16.2.1 NMEAData() [1/2]	149
8.16.2.2 NMEAData() [2/2]	150
8.16.3 Member Function Documentation	150
8.16.3.1 operator=()	150
8.16.3.2 get_instance()	150

8.16.3.3 update_rmc_tokens()	151
8.16.3.4 update_gga_tokens()	151
8.16.3.5 get_rmc_tokens()	152
8.16.3.6 get_gga_tokens()	152
8.16.4 Member Data Documentation	152
8.16.4.1 rmc_tokens_	152
8.16.4.2 gga_tokens_	153
8.16.4.3 rmc_mutex_	153
8.16.4.4 gga_mutex_	153
8.17 PowerManager Class Reference	153
8.17.1 Detailed Description	155
8.17.2 Constructor & Destructor Documentation	155
8.17.2.1 PowerManager() [1/2]	155
8.17.2.2 PowerManager() [2/2]	156
8.17.3 Member Function Documentation	156
8.17.3.1 operator=()	156
8.17.4 Member Data Documentation	156
8.17.4.1 SOLAR_CURRENT_THRESHOLD	156
8.17.4.2 USB_CURRENT_THRESHOLD	157
8.17.4.3 BATTERY_LOW_THRESHOLD	157
8.17.4.4 BATTERY_FULL_THRESHOLD	157
8.17.4.5 ina3221_	157
8.17.4.6 initialized_	157
8.17.4.7 powerman_mutex_	157
8.18 SensorDataRecord Struct Reference	158
8.18.1 Detailed Description	158
8.18.2 Member Data Documentation	158
8.18.2.1 timestamp	158
8.18.2.2 temperature	158
8.18.2.3 pressure	159
8.18.2.4 humidity	159
8.18.2.5 light	159
8.19 SensorWrapper Class Reference	159
8.19.1 Detailed Description	160
8.19.2 Constructor & Destructor Documentation	160
8.19.2.1 SensorWrapper()	160
8.19.3 Member Function Documentation	161
8.19.3.1 get_instance()	161
8.19.3.2 scan_connected_sensors()	161
8.19.3.3 get_available_sensors()	162
8.19.4 Member Data Documentation	162
8.19.4.1 sensors	162

8.20 SystemStateManager Class Reference	162
8.20.1 Detailed Description	164
8.20.2 Constructor & Destructor Documentation	164
8.20.2.1 SystemStateManager() [1/2]	164
8.20.2.2 SystemStateManager() [2/2]	165
8.20.3 Member Function Documentation	165
8.20.3.1 operator=()	165
8.20.3.2 get_instance()	165
8.20.3.3 is_bootloader_reset_pending()	166
8.20.3.4 set_bootloader_reset_pending()	167
8.20.3.5 is_gps_collection_paused()	167
8.20.3.6 set_gps_collection_paused()	167
8.20.3.7 is_sd_card_mounted()	168
8.20.3.8 set_sd_card_mounted()	168
8.20.3.9 get_uart_verbosity()	169
8.20.3.10 set_uart_verbosity()	169
8.20.3.11 is_radio_init_ok()	170
8.20.3.12 set_radio_init_ok()	170
8.20.3.13 is_light_sensor_init_ok()	171
8.20.3.14 set_light_sensor_init_ok()	171
8.20.3.15 is_env_sensor_init_ok()	171
8.20.3.16 set_env_sensor_init_ok()	171
8.20.3.17 get_operating_mode()	172
8.20.3.18 set_operating_mode()	172
8.20.4 Member Data Documentation	173
8.20.4.1 pending_bootloader_reset	173
8.20.4.2 gps_collection_paused	173
8.20.4.3 sd_card_mounted	173
8.20.4.4 uart_verbosity	173
8.20.4.5 sd_card_init_status	174
8.20.4.6 radio_init_status	174
8.20.4.7 light_sensor_init_status	174
8.20.4.8 env_sensor_init_status	174
8.20.4.9 system_operating_mode	174
8.20.4.10 mutex_	174
8.21 TelemetryManager Class Reference	175
8.21.1 Detailed Description	176
8.21.2 Constructor & Destructor Documentation	177
8.21.2.1 ~TelemetryManager()	177
8.21.3 Member Function Documentation	177
8.21.3.1 get_instance()	177
8.21.3.2 flush_sensor_data()	178

8.21.3.3 get_telemetry_buffer_count()	178
8.21.3.4 get_telemetry_buffer_write_index()	178
8.21.4 Member Data Documentation	178
8.21.4.1 TELEMETRY_BUFFER_SIZE	178
8.21.4.2 DEFAULT_SAMPLE_INTERVAL_MS	178
8.21.4.3 DEFAULT_FLUSH_THRESHOLD	178
8.21.4.4 sample_interval_ms	179
8.21.4.5 flush_threshold	179
8.21.4.6 telemetry_buffer	179
8.21.4.7 telemetry_buffer_count	179
8.21.4.8 telemetry_buffer_write_index	179
8.21.4.9 sensor_data_buffer	179
8.21.4.10 last_telemetry_record_copy	179
8.21.4.11 last_sensor_record_copy	180
8.21.4.12 telemetry_mutex	180
8.22 TelemetryRecord Struct Reference	180
8.22.1 Detailed Description	181
8.22.2 Member Data Documentation	181
8.22.2.1 timestamp	181
8.22.2.2 build_version	181
8.22.2.3 battery_voltage	181
8.22.2.4 system_voltage	181
8.22.2.5 charge_current_usb	181
8.22.2.6 charge_current_solar	182
8.22.2.7 discharge_current	182
8.22.2.8 time	182
8.22.2.9 latitude	182
8.22.2.10 lat_dir	182
8.22.2.11 longitude	182
8.22.2.12 lon_dir	183
8.22.2.13 speed	183
8.22.2.14 course	183
8.22.2.15 date	183
8.22.2.16 fix_quality	183
8.22.2.17 satellites	183
8.22.2.18 altitude	183
9 File Documentation	185
9.1 build_number.h File Reference	185
9.1.1 Macro Definition Documentation	185
9.1.1.1 BUILD_NUMBER	185
9.2 build_number.h	185

9.3 credits.md File Reference	186
9.4 includes.h File Reference	186
9.5 includes.h	187
9.6 lib/clock/DS3231.cpp File Reference	187
9.7 DS3231.cpp	188
9.8 lib/clock/DS3231.h File Reference	190
9.8.1 Macro Definition Documentation	191
9.8.1.1 DS3231_DEVICE_ADDRESS	191
9.8.1.2 DS3231_SECONDS_REG	191
9.8.1.3 DS3231_MINUTES_REG	192
9.8.1.4 DS3231_HOURS_REG	192
9.8.1.5 DS3231_DAY_REG	192
9.8.1.6 DS3231_DATE_REG	192
9.8.1.7 DS3231_MONTH_REG	192
9.8.1.8 DS3231_YEAR_REG	192
9.8.1.9 DS3231_CONTROL_REG	193
9.8.1.10 DS3231_CONTROL_STATUS_REG	193
9.8.1.11 DS3231_TEMPERATURE_MSB_REG	193
9.8.1.12 DS3231_TEMPERATURE_LSB_REG	193
9.8.2 Enumeration Type Documentation	193
9.8.2.1 days_of_week	193
9.9 DS3231.h	194
9.10 lib/comms/commands/clock_commands.cpp File Reference	195
9.10.1 Variable Documentation	195
9.10.1.1 clock_commands_group_id	195
9.10.1.2 time_command_id	196
9.10.1.3 timezone_offset_command_id	196
9.10.1.4 internal_temperature_command_id	196
9.11 clock_commands.cpp	196
9.12 lib/comms/commands/commands.cpp File Reference	198
9.13 commands.cpp	199
9.14 lib/comms/commands/commands.h File Reference	199
9.15 commands.h	201
9.16 lib/comms/commands/diagnostic_commands.cpp File Reference	202
9.17 diagnostic_commands.cpp	203
9.18 lib/comms/commands/event_commands.cpp File Reference	205
9.19 event_commands.cpp	206
9.20 lib/comms/commands/gps_commands.cpp File Reference	207
9.20.1 Variable Documentation	208
9.20.1.1 gps_commands_group_id	208
9.20.1.2 power_status_command_id	208
9.20.1.3 passthrough_command_id	208

9.21 gps_commands.cpp	208
9.22 lib/comms/commands/telemetry_commands.cpp File Reference	210
9.22.1 Variable Documentation	211
9.22.1.1 telemetry_commands_group	211
9.22.1.2 last_telemetry_command_id	211
9.22.1.3 last_sensor_command_id	211
9.23 telemetry_commands.cpp	211
9.24 lib/comms/communication.cpp File Reference	212
9.24.1 Function Documentation	213
9.24.1.1 initialize_radio()	213
9.24.1.2 lora_tx_done_callback()	213
9.25 communication.cpp	214
9.26 lib/comms/communication.h File Reference	214
9.26.1 Function Documentation	216
9.26.1.1 initialize_radio()	216
9.26.1.2 lora_tx_done_callback()	216
9.26.1.3 on_receive()	217
9.26.1.4 handle_uart_input()	218
9.26.1.5 send_message()	218
9.26.1.6 send_frame_uart()	219
9.26.1.7 send_frame_lora()	220
9.27 communication.h	220
9.28 lib/comms/frame.cpp File Reference	221
9.28.1 Detailed Description	221
9.28.2 Typedef Documentation	222
9.28.2.1 CommandHandler	222
9.29 frame.cpp	222
9.30 lib/comms/protocol.h File Reference	224
9.30.1 Variable Documentation	225
9.30.1.1 FRAME_BEGIN	225
9.30.1.2 FRAME_END	225
9.30.1.3 DELIMITER	226
9.31 protocol.h	226
9.32 lib/comms/receive.cpp File Reference	227
9.32.1 Macro Definition Documentation	227
9.32.1.1 MAX_PACKET_SIZE	227
9.32.2 Function Documentation	228
9.32.2.1 extract_and_process_frames()	228
9.32.2.2 process_lora_packet()	228
9.32.2.3 on_receive()	229
9.32.2.4 handle_uart_input()	230
9.33 receive.cpp	231

9.34 lib/comms/send.cpp File Reference	232
9.34.1 Detailed Description	233
9.34.2 Function Documentation	233
9.34.2.1 send_message()	233
9.34.2.2 send_frame_lora()	234
9.34.2.3 send_frame_uart()	234
9.35 send.cpp	235
9.36 lib/comms/utils_converters.cpp File Reference	235
9.36.1 Detailed Description	236
9.37 utils_converters.cpp	236
9.38 lib/eventman/event_manager.cpp File Reference	237
9.38.1 Detailed Description	237
9.39 event_manager.cpp	237
9.40 lib/eventman/event_manager.h File Reference	239
9.40.1 Detailed Description	241
9.40.2 Variable Documentation	241
9.40.2.1 timestamp	241
9.40.2.2 id	241
9.40.2.3 group	241
9.40.2.4 event	242
9.41 event_manager.h	242
9.42 lib/location/gps_collector.cpp File Reference	243
9.42.1 Detailed Description	244
9.43 gps_collector.cpp	244
9.44 lib/location/gps_collector.h File Reference	245
9.44.1 Detailed Description	246
9.45 gps_collector.h	247
9.46 lib/location/NMEA/NMEA_data.h File Reference	247
9.46.1 Detailed Description	248
9.47 NMEA_data.h	248
9.48 lib/pin_config.h File Reference	249
9.48.1 Macro Definition Documentation	250
9.48.1.1 DEBUG_UART_PORT	250
9.48.1.2 DEBUG_UART_BAUD_RATE	250
9.48.1.3 DEBUG_UART_TX_PIN	250
9.48.1.4 DEBUG_UART_RX_PIN	250
9.48.1.5 MAIN_I2C_PORT	251
9.48.1.6 MAIN_I2C_SDA_PIN	251
9.48.1.7 MAIN_I2C_SCL_PIN	251
9.48.1.8 GPS_UART_PORT	251
9.48.1.9 GPS_UART_BAUD_RATE	251
9.48.1.10 GPS_UART_TX_PIN	251

9.48.1.11 GPS_UART_RX_PIN	251
9.48.1.12 GPS_POWER_ENABLE_PIN	251
9.48.1.13 SENSORS_POWER_ENABLE_PIN	252
9.48.1.14 SENSORS_I2C_PORT	252
9.48.1.15 SENSORS_I2C_SDA_PIN	252
9.48.1.16 SENSORS_I2C_SCL_PIN	252
9.48.1.17 BUFFER_SIZE	252
9.48.1.18 SD_SPI_PORT	252
9.48.1.19 SD_MISO_PIN	252
9.48.1.20 SD_MOSI_PIN	252
9.48.1.21 SD_SCK_PIN	253
9.48.1.22 SD_CS_PIN	253
9.48.1.23 SD_CARD_DETECT_PIN	253
9.48.1.24 SX1278_MISO	253
9.48.1.25 SX1278_CS	253
9.48.1.26 SX1278_SCK	253
9.48.1.27 SX1278_MOSI	253
9.48.1.28 SPI_PORT	253
9.48.1.29 READ_BIT	254
9.48.1.30 LORA_DEFAULT_SPI	254
9.48.1.31 LORA_DEFAULT_SPI_FREQUENCY	254
9.48.1.32 LORA_DEFAULT_SS_PIN	254
9.48.1.33 LORA_DEFAULT_RESET_PIN	254
9.48.1.34 LORA_DEFAULT_DIO0_PIN	254
9.48.1.35 PA_OUTPUT_RFO_PIN	254
9.48.1.36 PA_OUTPUT_PA_BOOST_PIN	254
9.48.2 Variable Documentation	255
9.48.2.1 lora_cs_pin	255
9.48.2.2 lora_reset_pin	255
9.48.2.3 lora_irq_pin	255
9.48.2.4 lora_address_local	255
9.48.2.5 lora_address_remote	255
9.49 pin_config.h	256
9.50 lib/powerman/INA3221/INA3221.cpp File Reference	256
9.50.1 Detailed Description	257
9.51 INA3221.cpp	257
9.52 lib/powerman/INA3221/INA3221.h File Reference	259
9.52.1 Detailed Description	261
9.52.2 Enumeration Type Documentation	261
9.52.2.1 ina3221_addr_t	261
9.52.2.2 ina3221_ch_t	261
9.52.2.3 ina3221_reg_t	261

9.52.2.4 ina3221_avg_mode_t	262
9.52.2.5 ina3221_op_mode_t	262
9.52.3 Variable Documentation	263
9.52.3.1 INA3221_CH_NUM	263
9.52.3.2 SHUNT_VOLTAGE_LSB_UV	263
9.53 INA3221.h	263
9.54 lib/powerman/PowerManager.cpp File Reference	265
9.54.1 Detailed Description	265
9.55 PowerManager.cpp	266
9.56 lib/powerman/PowerManager.h File Reference	267
9.56.1 Detailed Description	268
9.57 PowerManager.h	268
9.58 lib/sensors/BH1750/BH1750.cpp File Reference	269
9.58.1 Detailed Description	269
9.59 BH1750.cpp	269
9.60 lib/sensors/BH1750/BH1750.h File Reference	270
9.60.1 Detailed Description	271
9.61 BH1750.h	272
9.62 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference	272
9.63 BH1750_WRAPPER.cpp	273
9.64 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference	274
9.65 BH1750_WRAPPER.h	275
9.66 lib/sensors/BME280/BME280.cpp File Reference	275
9.66.1 Detailed Description	276
9.67 BME280.cpp	276
9.68 lib/sensors/BME280/BME280.h File Reference	279
9.68.1 Detailed Description	280
9.69 BME280.h	280
9.70 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference	282
9.71 BME280_WRAPPER.cpp	282
9.72 lib/sensors/BME280/BME280_WRAPPER.h File Reference	283
9.73 BME280_WRAPPER.h	284
9.74 lib/sensors/ISensor.cpp File Reference	284
9.74.1 Detailed Description	285
9.75 ISensor.cpp	285
9.76 lib/sensors/ISensor.h File Reference	286
9.76.1 Detailed Description	287
9.77 ISensor.h	287
9.78 lib/storage/storage.cpp File Reference	288
9.78.1 Detailed Description	288
9.79 storage.cpp	288
9.80 lib/storage/storage.h File Reference	289

9.80.1 Detailed Description	290
9.81 storage.h	290
9.82 lib/system_state_manager.h File Reference	291
9.82.1 Detailed Description	291
9.83 system_state_manager.h	292
9.84 lib/telemetry/telemetry_manager.cpp File Reference	293
9.84.1 Detailed Description	294
9.85 telemetry_manager.cpp	294
9.86 lib/telemetry/telemetry_manager.h File Reference	298
9.86.1 Detailed Description	299
9.87 telemetry_manager.h	299
9.88 lib/utils.cpp File Reference	301
9.88.1 Detailed Description	302
9.88.2 Function Documentation	302
9.88.2.1 get_level_color()	302
9.88.2.2 get_level_prefix()	303
9.88.2.3 uart_print()	304
9.89 utils.cpp	306
9.90 lib/utils.h File Reference	306
9.90.1 Detailed Description	307
9.90.2 Macro Definition Documentation	308
9.90.2.1 ANSI_RED	308
9.90.2.2 ANSI_GREEN	308
9.90.2.3 ANSI_YELLOW	308
9.90.2.4 ANSI_BLUE	308
9.90.2.5 ANSI_RESET	308
9.90.3 Enumeration Type Documentation	308
9.90.3.1 VerbosityLevel	308
9.90.4 Function Documentation	309
9.90.4.1 uart_print()	309
9.91 utils.h	310
9.92 main.cpp File Reference	311
9.92.1 Macro Definition Documentation	311
9.92.1.1 LOG_FILENAME	311
9.92.2 Function Documentation	312
9.92.2.1 core1_entry()	312
9.92.2.2 init_pico_hw()	312
9.92.2.3 init_modules()	313
9.92.2.4 define_system_operating_mode()	314
9.92.2.5 main()	314
9.93 main.cpp	315

Chapter 1

credits

this software uses following external libraries

1.1 pico-vfs

Copyright 2024, Hiroyuki OYAMA. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The following source code has been written with reference to the [ARM Embed OS](#) source code; ARM Embed OS is licensed under the Apache 2.0 licence.

- [src/blockdevice/sd.c](#)

Folders containing files by third parties are listed below. Each folder must contain a README file specifying the licence for that file. The original licence text is contained in these source files.

- [vendor/ff15](#) - BSD-2-Clause, [Copyright ChaN](#)
- [vendor/littlefs](#) - BSD-3-Clause

1.2 pico-lora

MIT License

Copyright (c) 2021 Akshaya Bali

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Chapter 2

Clock Commands

Member `handle_enable_gps_uart_passthrough (const std::string ¶m, OperationType operationType)`

Command ID: 7.2

Member `handle_enter_bootloader_mode (const std::string ¶m, OperationType operationType)`

Command ID: 2

Member `handle_get_build_version (const std::string ¶m, OperationType operationType)`

Command ID: 1

Member `handle_get_commands_list (const std::string ¶m, OperationType operationType)`

Command ID: 0

Member `handle_get_event_count (const std::string ¶m, OperationType operationType)`

Command ID: 5.2

Member `handle_get_internal_temperature (const std::string ¶m, OperationType operationType)`

Command ID: 3.4

Member `handle_get_last_events (const std::string ¶m, OperationType operationType)`

Command ID: 5.1

Member `handle_get_last telemetry_record (const std::string ¶m, OperationType operationType)`

Command ID: 8.2

Member `handle_get_power_mode (const std::string ¶m, OperationType operationType)`

Command ID: 1.2

Member `handle_get_uptime (const std::string ¶m, OperationType operationType)`

Command ID: 1.3

Member `handle_gps_power_status (const std::string ¶m, OperationType operationType)`

Command ID: 7.1

Member `handle_time (const std::string ¶m, OperationType operationType)`

Command ID: 3.0

Member `handle_timezone_offset (const std::string ¶m, OperationType operationType)`

Command ID: 3.1

Member `handle_verbosity (const std::string ¶m, OperationType operationType)`

Command ID: 1.8

Chapter 3

Topic Index

3.1 Topics

Here is a list of all topics with brief descriptions:

Clock Management Commands	13
Command System	16
Diagnostic Commands	19
Event Commands	25
GPS Commands	28
Telemetry Buffer Commands	30
Frame Handling	32
Protocol	36
Utility Converters	39
RTC clock	42
Event Management	51
Location	58
INA3221 Power Monitor	61
Configuration Functions	61
Measurement Functions	66
Power Management	68
BH1750 Light Sensor	74
Constants	77
Types	78
Sensors	79
Storage	83
System State Manager	84
Telemetry Manager	85

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

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

BH1750	97
BME280	103
BME280CalibParam	114
INA3221::conf_reg_t	122
DS3231	124
ds3231_data_t	128
EventEmitter	130
EventLog	131
EventManager	133
Frame	137
INA3221	139
ISensor	143
BH1750Wrapper	99
BME280Wrapper	119
INA3221::masken_reg_t	146
NMEAData	148
PowerManager	153
SensorDataRecord	158
SensorWrapper	159
SystemStateManager	162
TelemetryManager	175
TelemetryRecord	180

Chapter 5

Class Index

5.1 Class List

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

BH1750	Class to interface with the BH1750 light sensor	97
BH1750Wrapper		99
BME280	Class to interface with the BME280 environmental sensor	103
BME280CalibParam	Structure to hold the BME280 calibration parameters	114
BME280Wrapper		119
INA3221::conf_reg_t	Configuration register bit fields	122
DS3231	Class for interfacing with the DS3231 real-time clock	124
ds3231_data_t	Structure to hold time and date information from DS3231	128
EventEmitter	Provides a simple interface for emitting events	130
EventLog	Structure for storing event log data	131
EventManager	Manages event logging and storage	133
Frame	Represents a communication frame used for data exchange	137
INA3221	INA3221 Triple-Channel Power Monitor driver class	139
ISensor	Abstract base class for sensors	143
INA3221::masken_reg_t	Mask/Enable register bit fields	146
NMEAData	Manages parsed NMEA sentences	148
PowerManager	Manages power-related functions	153
SensorDataRecord	Structure representing a single sensor data point	158
SensorWrapper	Manages a collection of sensors	159

SystemStateManager	
Manages the system state of the Kubisat firmware	162
TelemetryManager	
Manages the collection, storage, and retrieval of telemetry data	175
TelemetryRecord	
Structure representing a single telemetry data point	180

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

<code>build_number.h</code>	185
<code>includes.h</code>	186
<code>main.cpp</code>	311
<code>lib/pin_config.h</code>	249
<code>lib/system_state_manager.h</code> Manages the system state of the Kabisat firmware	291
<code>lib/utils.cpp</code> Implementation of utility functions for the Kabisat firmware	301
<code>lib/utils.h</code> Utility functions and definitions for the Kabisat firmware	306
<code>lib/clock/DS3231.cpp</code>	187
<code>lib/clock/DS3231.h</code>	190
<code>lib/comms/communication.cpp</code>	212
<code>lib/comms/communication.h</code>	214
<code>lib/comms/frame.cpp</code> Implements functions for encoding, decoding, building, and processing Frames	221
<code>lib/comms/protocol.h</code>	224
<code>lib/comms/receive.cpp</code>	227
<code>lib/comms/send.cpp</code> Implements functions for sending data, including LoRa messages and Frames	232
<code>lib/comms/utils_converters.cpp</code> Implements utility functions for converting between different data types	235
<code>lib/comms/commands/clock_commands.cpp</code>	195
<code>lib/comms/commands/commands.cpp</code>	198
<code>lib/comms/commands/commands.h</code>	199
<code>lib/comms/commands/diagnostic_commands.cpp</code>	202
<code>lib/comms/commands/event_commands.cpp</code>	205
<code>lib/comms/commands/gps_commands.cpp</code>	207
<code>lib/comms/commands/telemetry_commands.cpp</code>	210
<code>lib/eventman/event_manager.cpp</code> Implementation of the Event Manager and Event Emitter classes	237
<code>lib/eventman/event_manager.h</code> Header file for the Event Manager and Event Emitter classes	239
<code>lib/location/gps_collector.cpp</code> Implementation of the GPS data collector module	243

lib/location/gps_collector.h	
Header file for the GPS data collector module	245
lib/location/NMEA/NMEA_data.h	
Header file for the NMEAData class, which manages parsed NMEA sentences	247
lib/powerman/PowerManager.cpp	
Implementation of the PowerManager class, which manages power-related functions	265
lib/powerman/PowerManager.h	
Header file for the PowerManager class, which manages power-related functions	267
lib/powerman/INA3221/INA3221.cpp	
Implementation of the INA3221 power monitor driver	256
lib/powerman/INA3221/INA3221.h	
Header file for the INA3221 triple-channel power monitor driver	259
lib/sensors/ISensor.cpp	
Implementation of the ISensor interface and SensorWrapper class	284
lib/sensors/ISensor.h	
Header file for the ISensor interface and SensorWrapper class	286
lib/sensors/BH1750/BH1750.cpp	
Implementation of the BH1750 light sensor class	269
lib/sensors/BH1750/BH1750.h	
Header file for the BH1750 light sensor class	270
lib/sensors/BH1750/BH1750_WRAPPER.cpp	
.	272
lib/sensors/BH1750/BH1750_WRAPPER.h	
.	274
lib/sensors/BME280/BME280.cpp	
Implementation of the BME280 environmental sensor class	275
lib/sensors/BME280/BME280.h	
Header file for the BME280 environmental sensor class	279
lib/sensors/BME280/BME280_WRAPPER.cpp	
.	282
lib/sensors/BME280/BME280_WRAPPER.h	
.	283
lib/storage/storage.cpp	
Implements file system operations for the Kubisat firmware	288
lib/storage/storage.h	
Header file for file system operations on the Kubisat firmware	289
lib/telemetry/telemetry_manager.cpp	
Implementation of telemetry collection and storage functionality	293
lib/telemetry/telemetry_manager.h	
System telemetry collection and logging	298

Chapter 7

Topic Documentation

7.1 Clock Management Commands

Commands for managing system time and clock settings.

Functions

- std::vector< [Frame](#) > [handle_time](#) (const std::string ¶m, [OperationType](#) operationType)
Handler for getting and setting system time.
- std::vector< [Frame](#) > [handle_timezone_offset](#) (const std::string ¶m, [OperationType](#) operationType)
Handler for getting and setting timezone offset.
- std::vector< [Frame](#) > [handle_get_internal_temperature](#) (const std::string ¶m, [OperationType](#) operationType)
Handler for reading the [DS3231](#)'s internal temperature sensor.

7.1.1 Detailed Description

Commands for managing system time and clock settings.

7.1.2 Function Documentation

7.1.2.1 [handle_time\(\)](#)

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

Handler for getting and setting system time.

Parameters

<i>param</i>	For SET: Unix timestamp as string, for GET: empty string
<i>operationType</i>	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"

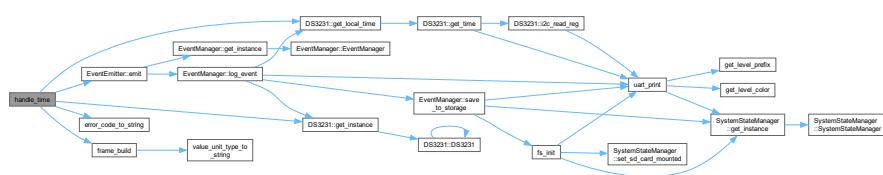
SET: KBST;0;SET;3;0;TIMESTAMP;KBST

When setting time, expects Unix timestamp as parameter

Command Command ID: 3.0

Definition at line 29 of file [clock_commands.cpp](#).

Here is the call graph for this function:



7.1.2.2 handle_timezone_offset()

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

Handler for getting and setting timezone offset.

Parameters

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

Returns

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

Note

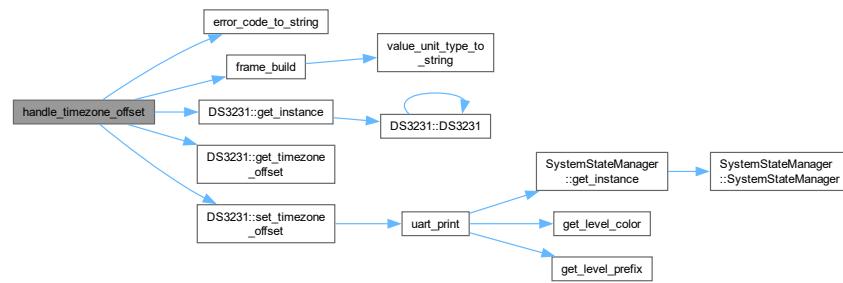
GET: **KBST;0;GET;3;1;;KBST**

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

Command Command ID: 3.1

Definition at line 94 of file [clock_commands.cpp](#).

Here is the call graph for this function:

**7.1.2.3 handle_get_internal_temperature()**

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

Handler for reading the **DS3231**'s internal temperature sensor.

Parameters

<i>param</i>	Empty string expected
<i>operationType</i>	GET

Returns

Vector with frame containing success/error and temperature in Celsius

Note

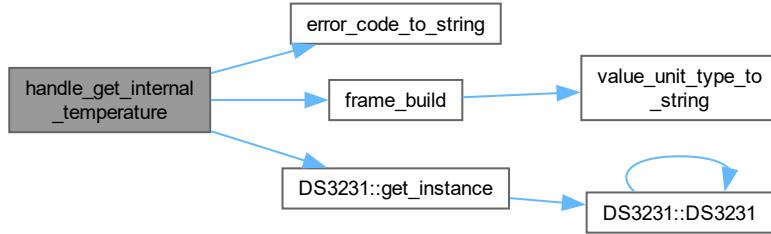
GET: **KBST;0;GET;3;4;;KBST**

Returns temperature in format "XX.XX" where XX.XX is temperature in Celsius

Command Command ID: 3.4

Definition at line 153 of file [clock_commands.cpp](#).

Here is the call graph for this function:



7.2 Command System

Core command system implementation.

Macros

- `#define CMD(group, cmd)`

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 ¶m, OperationType operationType)`
Executes a command based on its key.

Variables

- `CommandMap command_handlers`
Global map of all command handlers.

7.2.1 Detailed Description

Core command system implementation.

7.2.2 Macro Definition Documentation

7.2.2.1 CMD

```
#define CMD( group, cmd) ((static_cast<uint32_t>(group) << 8) | static_cast<uint32_t>(cmd))
```

Value:

```
((static_cast<uint32_t>(group) << 8) | static_cast<uint32_t>(cmd))
```

Definition at line 10 of file [commands.cpp](#).

7.2.3 Typedef Documentation

7.2.3.1 CommandHandler

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

Function type for command handlers.

Definition at line 16 of file [commands.cpp](#).

7.2.3.2 CommandMap

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

Map type for storing command handlers.

Definition at line 22 of file [commands.cpp](#).

7.2.4 Function Documentation

7.2.4.1 execute_command()

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

Executes a command based on its key.

Parameters

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

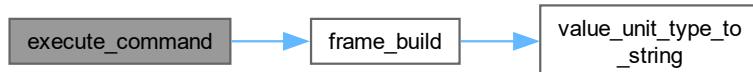
Returns

Frame Response frame containing execution result

Looks up the command handler in commandHandlers map and executes it

Definition at line 59 of file [commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.2.5 Variable Documentation

7.2.5.1 command_handlers

[CommandMap](#) command_handlers

Initial value:

```

= {
  {CMD(1, 0), handle_get_commands_list},
  {CMD(1, 1), handle_get_build_version},
  {CMD(1, 2), handle_get_power_mode},
  {CMD(1, 3), handle_get_uptime},
  {CMD(1, 8), handle_verbosity},
  {CMD(1, 9), handle_enter_bootloader_mode},

  {CMD(3, 0), handle_time},
  {CMD(3, 1), handle_timezone_offset},
  {CMD(3, 4), handle_get_internal_temperature},

  {CMD(5, 1), handle_get_last_events},
  {CMD(5, 2), handle_get_event_count},

  {CMD(7, 1), handle_gps_power_status},
  {CMD(7, 2), handle_enable_gps_uart_passthrough},

  {CMD(8, 2), handle_get_last_telemetry_record},
  {CMD(8, 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 28 of file [commands.cpp](#).

7.3 Diagnostic Commands

Functions

- std::vector< Frame > handle_get_commands_list (const std::string ¶m, OperationType operationType)
Handler for listing all available commands on UART.
- std::vector< Frame > handle_get_build_version (const std::string ¶m, OperationType operationType)
Get firmware build version.
- std::vector< Frame > handle_get_uptime (const std::string ¶m, OperationType operationType)
Get system uptime.
- std::vector< Frame > handle_get_power_mode (const std::string ¶m, OperationType operationType)
Get system power mode.
- std::vector< Frame > handle_verbosity (const std::string ¶m, OperationType operationType)
Handles setting or getting the UART verbosity level.
- std::vector< Frame > handle_enter_bootloader_mode (const std::string ¶m, OperationType operationType)
Reboot system to USB firmware loader.

Variables

- static constexpr uint8_t diagnostic_commands_group_id = 1
- static constexpr uint8_t commands_list_command_id = 0
- static constexpr uint8_t build_version_command_id = 1
- static constexpr uint8_t power_mode_command_id = 2
- static constexpr uint8_t uptime_command_id = 3
- static constexpr uint8_t verbosity_command_id = 8
- static constexpr uint8_t enter_bootloader_command_id = 9

7.3.1 Detailed Description

7.3.2 Function Documentation

7.3.2.1 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

<i>param</i>	Empty string expected
<i>operationType</i>	GET

Returns

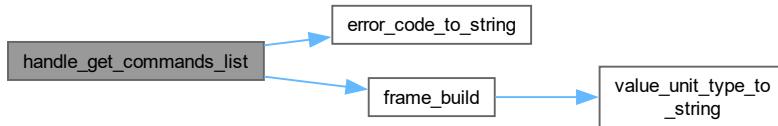
Vector of response frames - start frame, sequence of elements, end frame

Note**KBST;0;GET;1;0;;TSBK**

Print all available commands on UART port

Command Command ID: 0Definition at line 29 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:

**7.3.2.2 handle_get_build_version()**

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

Get firmware build version.

Parameters

<i>param</i>	Empty string expected
<i>operationType</i>	GET

Returns

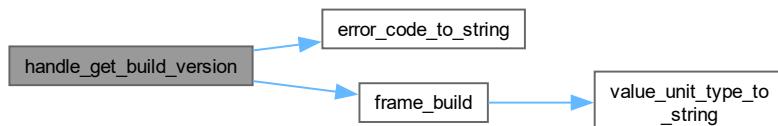
One-element vector with result frame

Note**KBST;0;GET;1;1;;TSBK**

Get the firmware build version

Command Command ID: 1Definition at line 83 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



7.3.2.3 handle_get_uptime()

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

Get system uptime.

Parameters

<i>param</i>	Empty string expected
<i>operationType</i>	GET

Returns

One-element vector with result frame

Note

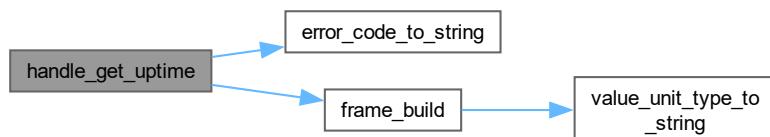
KBST;0;GET;1;3;;TSBK

Get the system uptime in seconds

Command Command ID: 1.3

Definition at line 114 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



7.3.2.4 handle_get_power_mode()

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

Get system power mode.

Parameters

<i>param</i>	Empty string expected
<i>operationType</i>	GET

Returns

One-element vector with result frame

Note

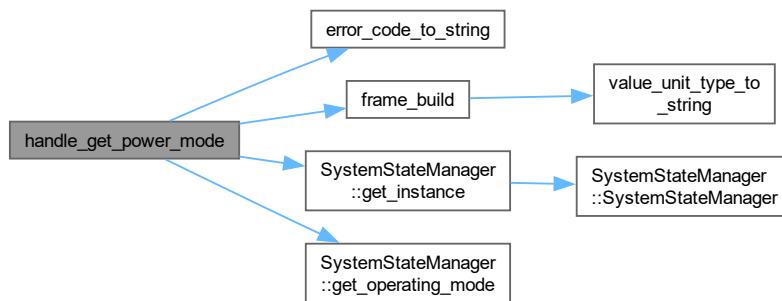
KBST;0;GET;1;2;;TSBK

Get the system power mode (BATTERY or USB)

Command Command ID: 1.2

Definition at line 146 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:

**7.3.2.5 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

<code>param</code>	The desired verbosity level (0-5) as a string. If empty, the current level is returned.
<code>operationType</code>	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.

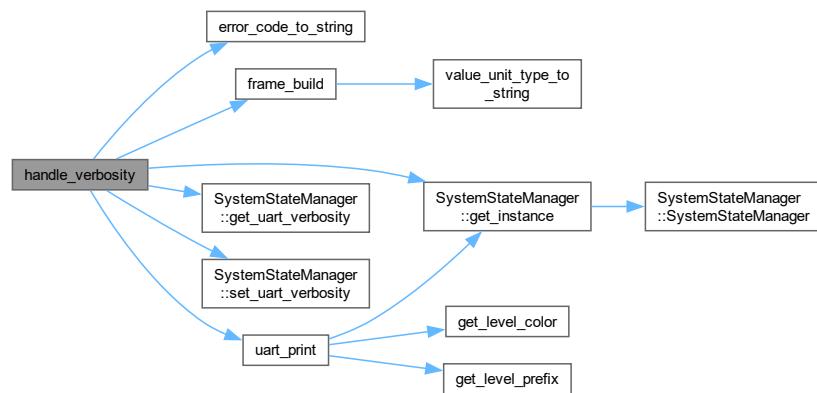
KBST;0;SET;1;8;[level];TSBK - Sets the verbosity level.

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

Command Command ID: 1.8

Definition at line 190 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



7.3.2.6 handle_enter_bootloader_mode()

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

Reboot system to USB firmware loader.

Parameters

<i>param</i>	Empty string expected
<i>operationType</i>	Must be SET

Returns

`Frame` with operation result

Note

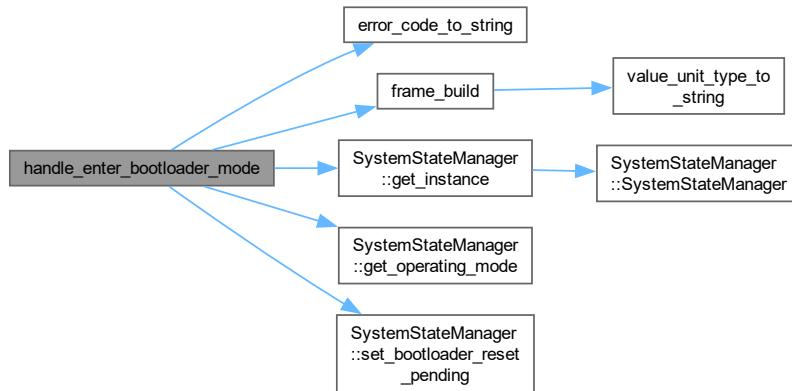
KBST;0;SET;1;9;;TSBK

Reboot the system to USB firmware loader

Command Command ID: 2

Definition at line 243 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



7.3.3 Variable Documentation

7.3.3.1 diagnostic_commands_group_id

```
uint8_t diagnostic_commands_group_id = 1 [static], [constexpr]
```

Definition at line 11 of file [diagnostic_commands.cpp](#).

7.3.3.2 commands_list_command_id

```
uint8_t commands_list_command_id = 0 [static], [constexpr]
```

Definition at line 12 of file [diagnostic_commands.cpp](#).

7.3.3.3 build_version_command_id

```
uint8_t build_version_command_id = 1 [static], [constexpr]
```

Definition at line 13 of file [diagnostic_commands.cpp](#).

7.3.3.4 power_mode_command_id

```
uint8_t power_mode_command_id = 2 [static], [constexpr]
```

Definition at line 14 of file [diagnostic_commands.cpp](#).

7.3.3.5 uptime_command_id

```
uint8_t uptime_command_id = 3 [static], [constexpr]
```

Definition at line 15 of file [diagnostic_commands.cpp](#).

7.3.3.6 verbosity_command_id

```
uint8_t verbosity_command_id = 8 [static], [constexpr]
```

Definition at line 16 of file [diagnostic_commands.cpp](#).

7.3.3.7 enter_bootloader_command_id

```
uint8_t enter_bootloader_command_id = 9 [static], [constexpr]
```

Definition at line 17 of file [diagnostic_commands.cpp](#).

7.4 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.

Variables

- static constexpr uint8_t [event_commands_group_id](#) = 5
- static constexpr uint8_t [last_events_command_id](#) = 1
- static constexpr uint8_t [event_count_command_id](#) = 2

7.4.1 Detailed Description

Commands for accessing and managing system event logs.

7.4.2 Function Documentation

7.4.2.1 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

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

Returns

Frame containing:

- Success: A sequence of frames, each containing up to 10 hex-encoded events. Each event is in the format IIIITTTTTTGGEE, 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

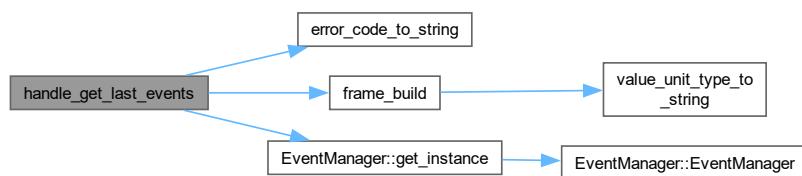
KBST;0;GET;5;1;[N];TSBK - Retrieves the last N events. If N is 0, retrieves all events.

Returns up to 10 most recent events per frame.

Command Command ID: 5.1

Definition at line 38 of file [event_commands.cpp](#).

Here is the call graph for this function:



7.4.2.2 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

<i>param</i>	Empty string expected
<i>operationType</i>	GET

Returns

Frame containing:

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

Note

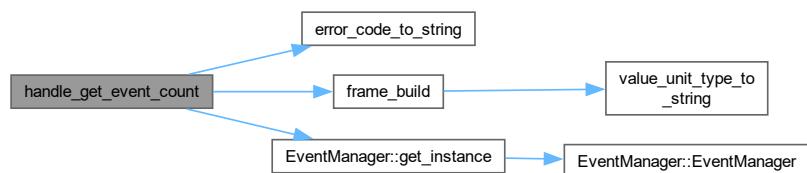
KBST;0;GET;5;2;;TSBK

Returns the total number of events in the log

Command Command ID: 5.2

Definition at line 106 of file [event_commands.cpp](#).

Here is the call graph for this function:



7.4.3 Variable Documentation

7.4.3.1 event_commands_group_id

```
uint8_t event_commands_group_id = 5 [static], [constexpr]
```

Definition at line 12 of file [event_commands.cpp](#).

7.4.3.2 last_events_command_id

```
uint8_t last_events_command_id = 1 [static], [constexpr]
```

Definition at line 13 of file [event_commands.cpp](#).

7.4.3.3 event_count_command_id

```
uint8_t event_count_command_id = 2 [static], [constexpr]
```

Definition at line 14 of file [event_commands.cpp](#).

7.5 GPS Commands

Commands for controlling and monitoring the GPS module.

Functions

- std::vector< [Frame](#) > [handle_gps_power_status](#) (const std::string ¶m, [OperationType](#) operationType)
Handler for controlling GPS module power state.
- std::vector< [Frame](#) > [handle_enable_gps_uart_passthrough](#) (const std::string ¶m, [OperationType](#) operationType)
Handler for enabling GPS transparent mode (UART pass-through)

7.5.1 Detailed Description

Commands for controlling and monitoring the GPS module.

7.5.2 Function Documentation

7.5.2.1 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

<i>param</i>	For SET: "0" to power off, "1" to power on. For GET: empty
<i>operationType</i>	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

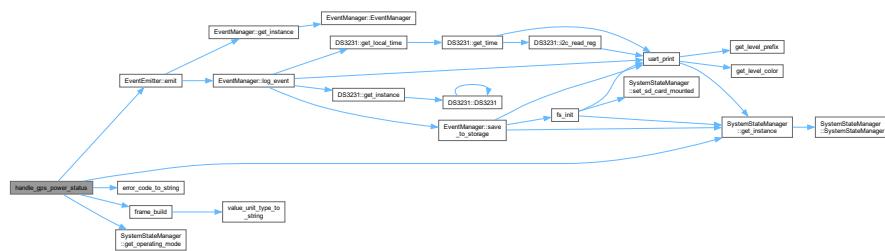
KBST;0;SET;7;1;POWER;TSBK

POWER - 0 - OFF, 1 - ON

Command Command ID: 7.1

Definition at line 31 of file [gps_commands.cpp](#).

Here is the call graph for this function:



7.5.2.2 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

<i>param</i>	TIMEOUT in seconds (optional, defaults to 60)
<i>operationType</i>	SET

Returns

Vector of Frames containing:

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

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

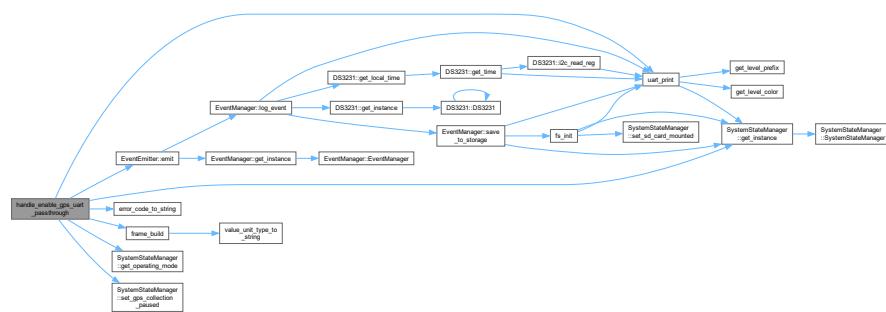
TIMEOUT - 1-600s, default 60s

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

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

Command Command ID: 7.2Definition at line 100 of file [gps_commands.cpp](#).

Here is the call graph for this function:



7.6 Telemetry Buffer Commands

Commands for interacting with the telemetry buffer.

Functions

- `std::vector< Frame > handle_get_last_telemetry_record (const std::string ¶m, OperationType operationType)`

Handles the get last record command.
- `std::vector< Frame > handle_get_last_sensor_record (const std::string ¶m, OperationType operationType)`

Handles the get last sensor record command.

7.6.1 Detailed Description

Commands for interacting with the telemetry buffer.

7.6.2 Function Documentation

7.6.2.1 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

<i>param</i>	Unused.
<i>operationType</i>	The operation type (must be GET).

Returns

A vector of Frames indicating the result of the operation.

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

Note

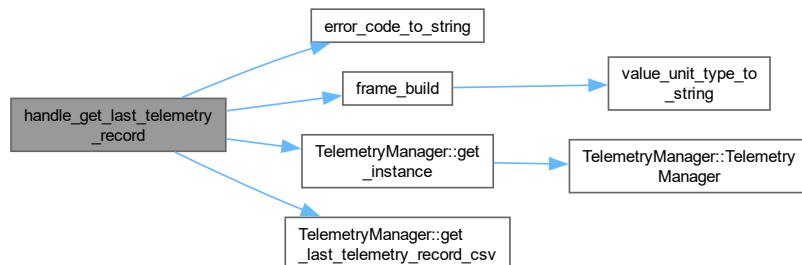
KBST;0;GET;8;2;;TSBK

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

Command Command ID: 8.2

Definition at line 32 of file [telemetry_commands.cpp](#).

Here is the call graph for this function:

**7.6.2.2 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

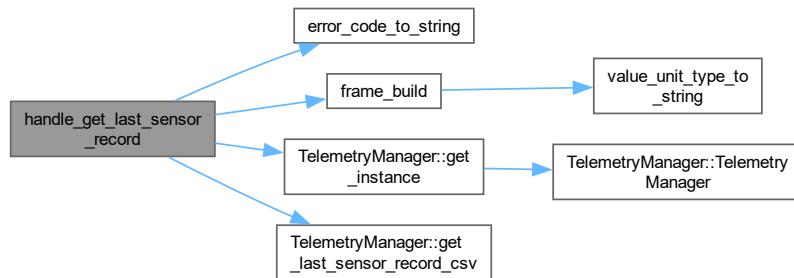
<i>param</i>	Unused.
<i>operationType</i>	The operation type (must be GET).

Returns

A vector of Frames indicating the result of the operation.

Definition at line 71 of file [telemetry_commands.cpp](#).

Here is the call graph for this function:



7.7 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.

7.7.1 Detailed Description

Functions for encoding, decoding and building communication frames.

7.7.2 Function Documentation

7.7.2.1 `frame_encode()`

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

Encodes a `Frame` instance into a string.

Parameters

<code>frame</code>	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;

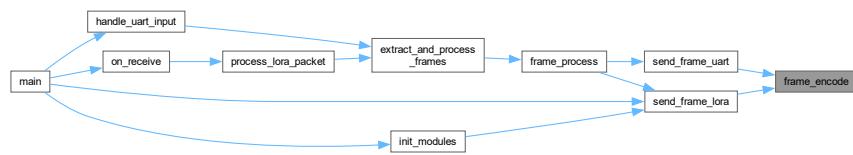
std::string encoded = frame_encode(myFrame);
// encoded will be "KBST;0;GET;1;1;;TSBK"
```

Definition at line 36 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

7.7.2.2 `frame_decode()`

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

Decodes a string into a [Frame](#) instance.

Parameters

<code>encodedFrame</code>	The string to decode.
---------------------------	-----------------------

Returns

The [Frame](#) instance decoded from the string.

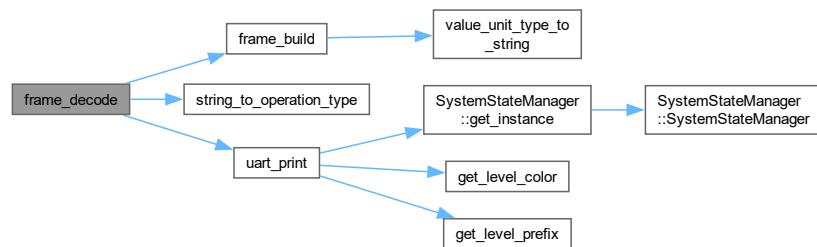
Exceptions

<code>std::runtime_error</code>	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 61 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.2.3 `frame_process()`

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

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

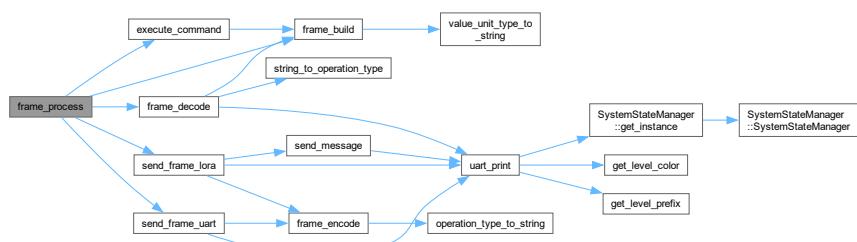
Parameters

<code>data</code>	The Frame data in string format.
-------------------	--

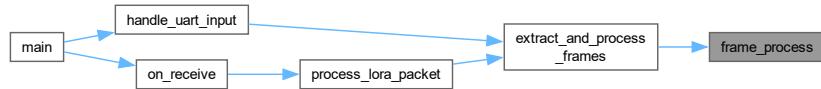
Decodes the frame data, extracts the command key, and executes the corresponding command. Sends the response frame. If an error occurs, an error frame is built and sent.

Definition at line 131 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.2.4 `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

<code>result</code>	The execution result.
<code>group</code>	The group ID.
<code>command</code>	The command ID within the group.
<code>value</code>	The payload value.
<code>unit</code>	The unit of measurement for the payload value.

Returns

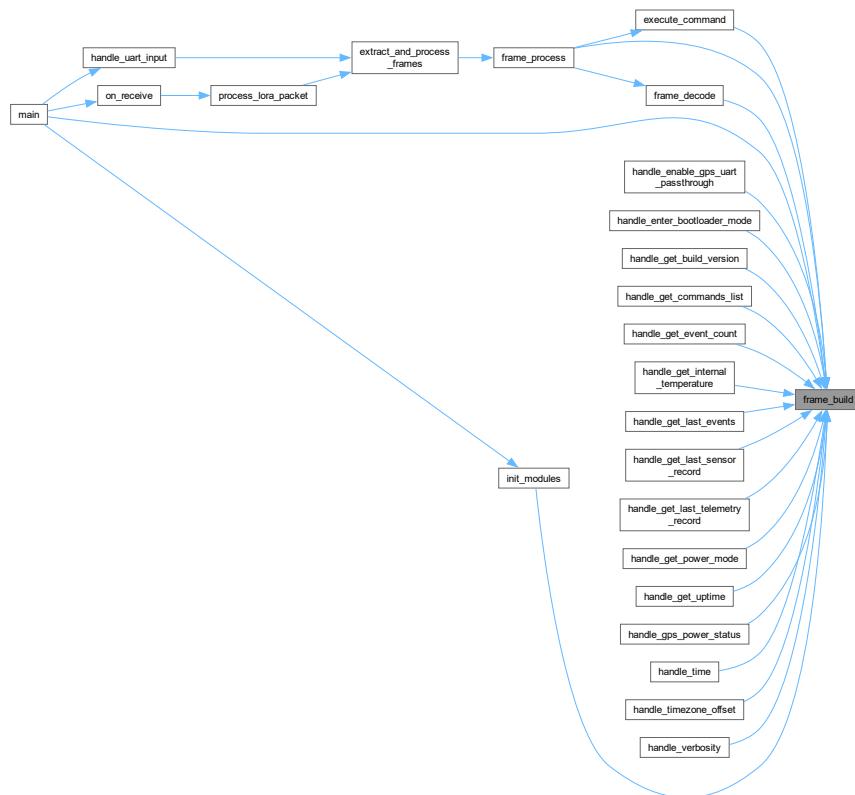
The [Frame](#) instance.

Definition at line 171 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.8 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` : `uint8_t` {
 `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` : `uint8_t` {
 `OperationType::GET` , `OperationType::SET` , `OperationType::RES` , `OperationType::VAL` ,
 `OperationType::SEQ` , `OperationType::ERR` }

Represents the type of operation being performed.
- enum class `ValueUnit` : `uint8_t` {
 `ValueUnit::UNDEFINED` , `ValueUnit::SECOND` , `ValueUnit::VOLT` , `ValueUnit::BOOL` ,
 `ValueUnit::DATETIME` , `ValueUnit::TEXT` , `ValueUnit::MILIAMP` , `ValueUnit::CELSIUS` }

Represents the unit of measurement for a payload value.
- enum class `Interface` : `uint8_t` { `Interface::UART` , `Interface::LORA` }

Represents the communication interface being used.

7.8.1 Detailed Description

Definitions for the communication protocol used by the satellite.

7.8.2 Enumeration Type Documentation

7.8.2.1 ErrorCode

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

Standard error codes for command responses.

Enumerator

<code>PARAM_UNNECESSARY</code>	
<code>PARAM_REQUIRED</code>	
<code>PARAM_INVALID</code>	
<code>INVALID_OPERATION</code>	
<code>NOT_ALLOWED</code>	
<code>INVALID_FORMAT</code>	
<code>INVALID_VALUE</code>	
<code>FAIL_TO_SET</code>	
<code>INTERNAL_FAIL_TO_READ</code>	
<code>UNKNOWN_ERROR</code>	

Definition at line 53 of file `protocol.h`.

7.8.2.2 OperationType

```
enum class OperationType : uint8_t [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](#).

7.8.2.3 ValueUnit

```
enum class ValueUnit : uint8_t [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.
CELSIUS	Unit is degrees Celsius.

Definition at line 94 of file [protocol.h](#).

7.8.2.4 Interface

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

Represents the communication interface being used.

Enumerator

UART	UART interface.
LORA	LoRa interface.

Definition at line 119 of file [protocol.h](#).

7.9 Utility Converters

Functions

- 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.

7.9.1 Detailed Description

7.9.2 Function Documentation

7.9.2.1 `value_unit_type_to_string()`

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

Converts a ValueUnit to a string.

Parameters

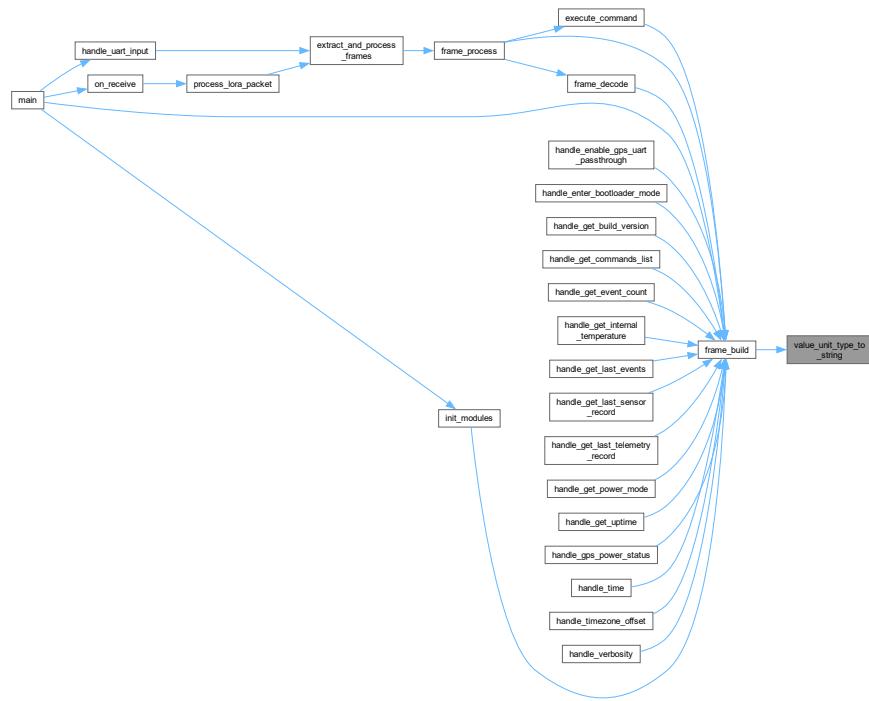
<code>unit</code>	The ValueUnit to convert.
-------------------	---------------------------

Returns

The string representation of the ValueUnit.

Definition at line 17 of file `utils_converters.cpp`.

Here is the caller graph for this function:



7.9.2.2 operation_type_to_string()

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

Converts an [OperationType](#) to a string.

Parameters

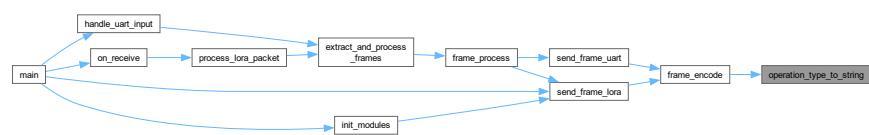
<code>type</code>	The OperationType to convert.
-------------------	---

Returns

The string representation of the [OperationType](#).

Definition at line 38 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



7.9.2.3 string_to_operation_type()

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

Converts a string to an [OperationType](#).

Parameters

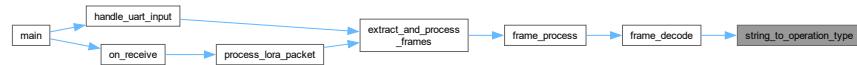
<code>str</code>	The string to convert.
------------------	------------------------

Returns

The [OperationType](#) corresponding to the string. Defaults to GET if the string is not recognized.

Definition at line 57 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



7.9.2.4 error_code_to_string()

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

Converts an [ErrorCode](#) to its string representation.

Parameters

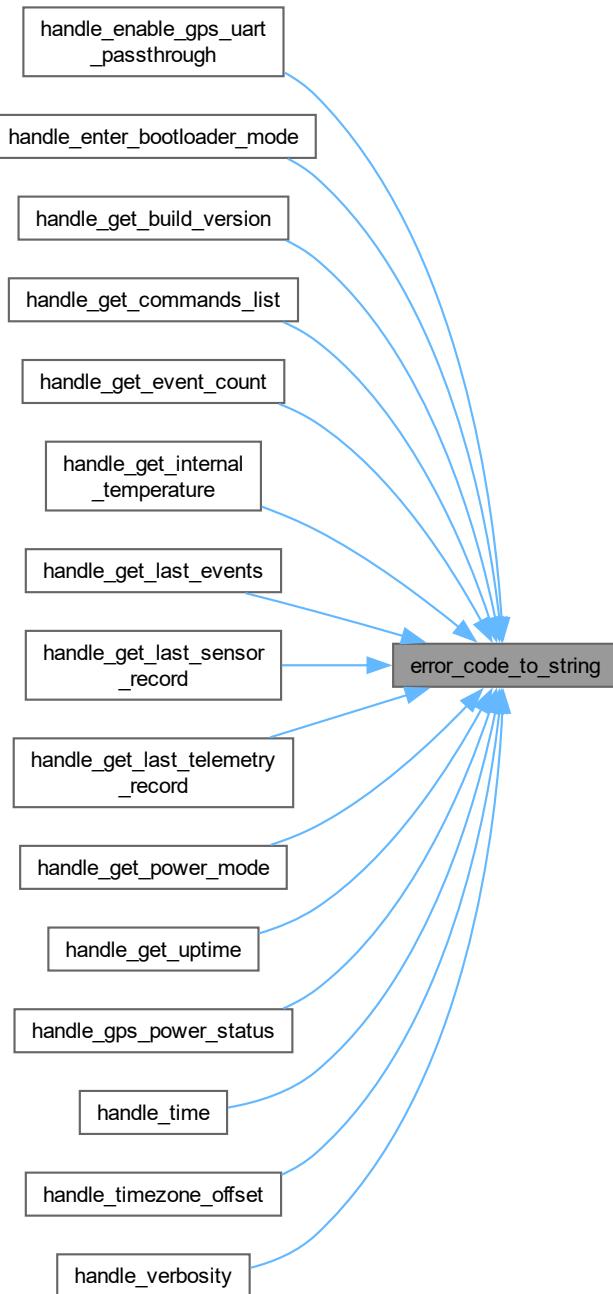
<code>code</code>	The error code
-------------------	----------------

Returns

String representation of the error code

Definition at line 73 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



7.10 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.
- `time_t DS3231::get_time ()`
Gets the current RTC time as Unix timestamp.
- `int DS3231::set_time (time_t unix_time)`
Sets the RTC time using a Unix timestamp.
- `int DS3231::read_temperature (float *resolution)`
Reads the current temperature from the DS3231.
- `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.
- `time_t DS3231::get_local_time ()`
Gets the current local time (including timezone offset)
- `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.

7.10.1 Detailed Description

Functions for interfacing with the DS3231 RTC module.

7.10.2 Function Documentation

7.10.2.1 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_ADDRESS.

Definition at line 23 of file `DS3231.cpp`.

7.10.2.2 `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

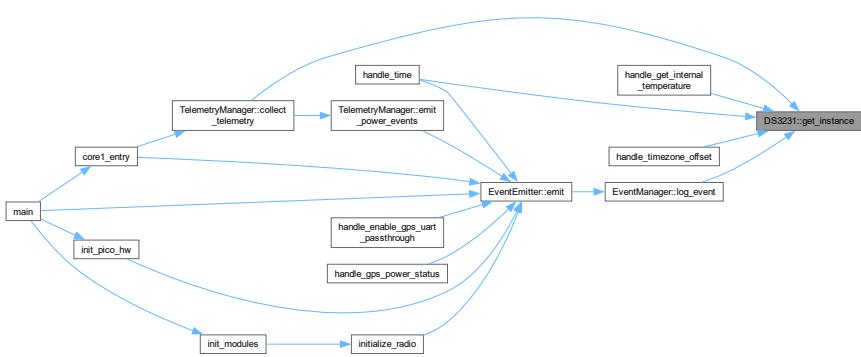
This function provides access to the single instance of the `DS3231` class, ensuring that only one object manages the RTC module. The instance is created upon the first call to this function and remains available for the lifetime of the program.

Definition at line 38 of file `DS3231.cpp`.

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.2.3 get_time()

```
time_t DS3231::get_time ()
```

Gets the current RTC time as Unix timestamp.

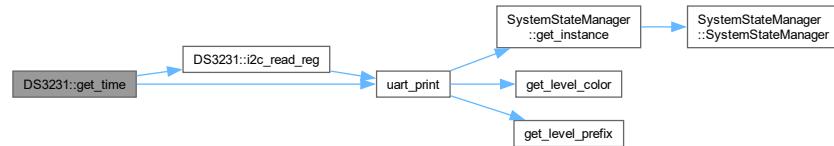
Gets current RTC time as Unix timestamp.

Returns

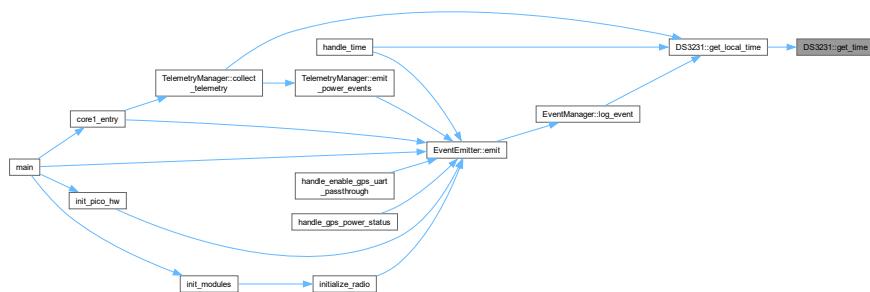
Unix timestamp or -1 on error

Definition at line 47 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.2.4 set_time()

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

Sets the RTC time using a Unix timestamp.

Sets the RTC time using Unix timestamp.

Parameters

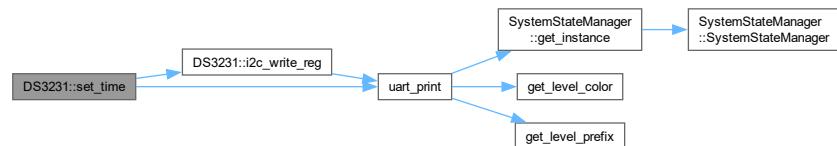
<i>unix_time</i>	Time in seconds since Unix epoch
------------------	----------------------------------

Returns

0 on success, -1 on failure

Definition at line 79 of file [DS3231.cpp](#).

Here is the call graph for this function:

**7.10.2.5 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

<code>out</code>	<code>resolution</code>	Pointer to store the temperature value in Celsius
------------------	-------------------------	---

Returns

0 on success, -1 on failure

Parameters

<code>out</code>	<code>resolution</code>	Pointer to a float to store the temperature value
------------------	-------------------------	---

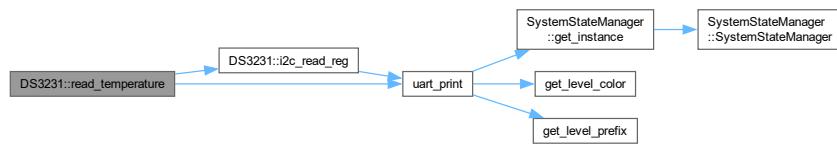
Returns

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 118 of file [DS3231.cpp](#).

Here is the call graph for this function:



7.10.2.6 get_timezone_offset()

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

Gets the current timezone offset.

Gets the currently configured timezone offset.

Returns

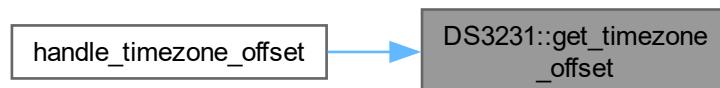
Timezone offset in minutes (-720 to +720)

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 146 of file [DS3231.cpp](#).

Here is the caller graph for this function:



7.10.2.7 set_timezone_offset()

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

Sets the timezone offset.

Parameters

	<code>offset_minutes</code>	Offset in minutes (-720 to +720)
in	<code>offset_minutes</code>	The timezone offset in minutes

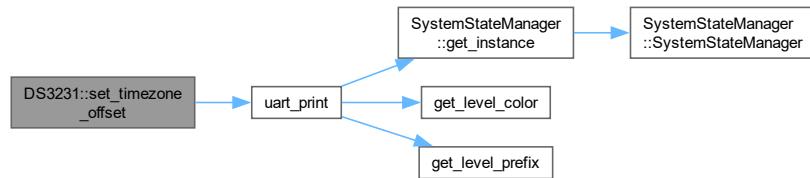
Sets the timezone offset in minutes relative to UTC. This value is used when converting between UTC and local time. The function validates that the offset is within a valid range (-720 to +720 minutes, which corresponds to -12 to +12 hours).

Note

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

Definition at line 162 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.2.8 `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

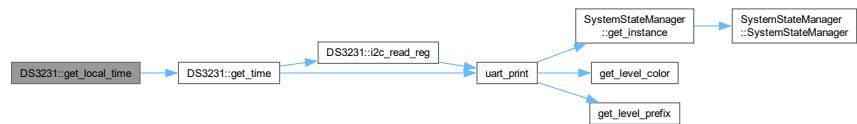
Unix timestamp adjusted for timezone, or -1 on error

Local time as Unix timestamp, or -1 on error

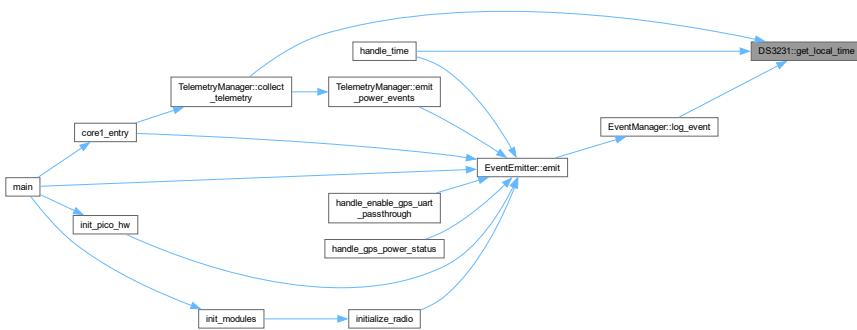
Retrieves the current UTC time from the RTC and applies the configured timezone offset (in minutes) to calculate the local time.

Definition at line 179 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.2.9 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	<i>reg_addr</i>	Register address to read from
in	<i>length</i>	Number of bytes to read
out	<i>data</i>	Buffer to store read data

Returns

0 on success, -1 on failure

Parameters

in	<i>reg_addr</i>	Register address to read from
in	<i>length</i>	Number of bytes to read
out	<i>data</i>	Buffer to store read data

Returns

0 on success, -1 on failure

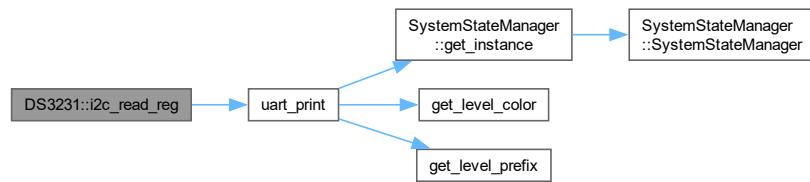
This method performs a thread-safe I²C read operation from the [DS3231](#). It first writes the register address to the device, then reads the requested number of bytes. All access is protected by a mutex to prevent concurrent I²C operations that could corrupt data.

Note

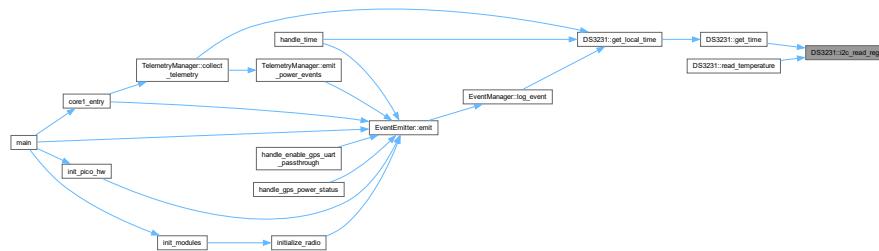
This is a low-level method used internally by the class.

Definition at line 206 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.10.2.10 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	<i>reg_addr</i>	Register address to write to
in	<i>length</i>	Number of bytes to write
in	<i>data</i>	Buffer containing data to write

Returns

0 on success, -1 on failure

Parameters

in	<i>reg_addr</i>	Register address to write to
in	<i>length</i>	Number of bytes to write
in	<i>data</i>	Buffer containing data to write

Returns

0 on success, -1 on failure

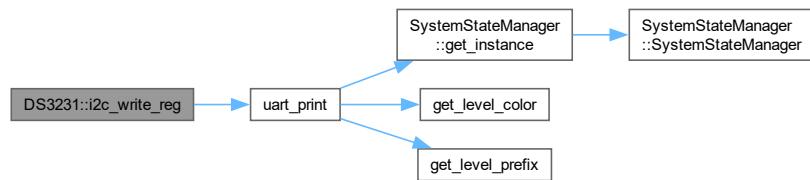
This method performs a thread-safe I²C write operation to the [DS3231](#). It combines the register address and data into a single buffer and sends it to the device. All access is protected by a mutex to prevent concurrent I²C operations that could corrupt data.

Note

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

Definition at line [247](#) of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.11 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` ,
`PowerEvent::DISCHARGING = 0x09` , `PowerEvent::CHARGING = 0x0A` }
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__`

7.11.1 Detailed Description

Classes and enums for handling system events.

7.11.2 Macro Definition Documentation

7.11.2.1 EVENT_BUFFER_SIZE

```
#define EVENT_BUFFER_SIZE 100
```

Size of the event buffer.

Definition at line 32 of file `event_manager.h`.

7.11.2.2 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`.

7.11.2.3 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`.

7.11.3 Enumeration Type Documentation

7.11.3.1 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](#).

7.11.3.2 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](#).

7.11.3.3 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.
DISCHARGING	Current balance negative.
CHARGING	Current balance positive.

Definition at line 87 of file [event_manager.h](#).

7.11.3.4 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 116 of file [event_manager.h](#).

7.11.3.5 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 134 of file [event_manager.h](#).

7.11.3.6 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 158 of file [event_manager.h](#).

7.11.4 Function Documentation

7.11.4.1 __attribute__()

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

7.11.4.2 init()

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

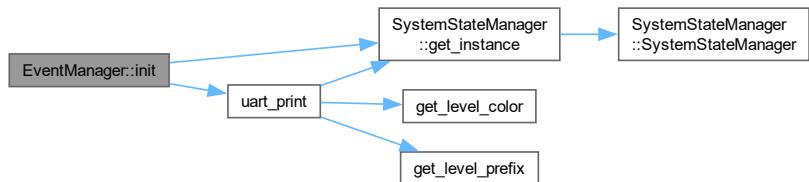
Initializes the event manager.

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:



7.11.4.3 log_event()

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

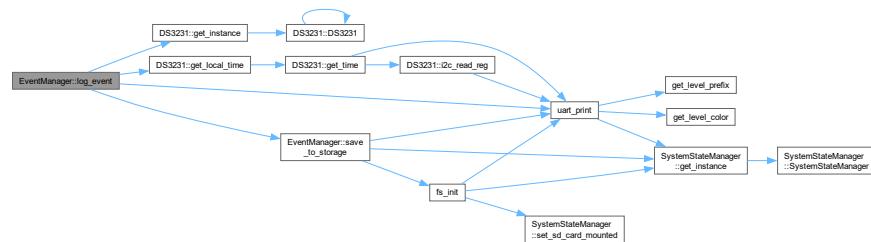
Logs an event to the event buffer.

Parameters

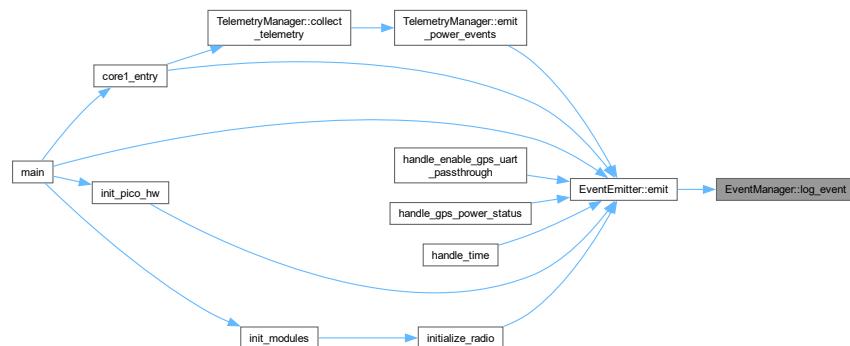
in	<i>group</i>	Event group.
in	<i>event</i>	Event code.

Definition at line 63 of file [event_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.11.4.4 `get_event()`

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

Gets an event from the event buffer.

Parameters

in	<i>index</i>	Index of the event to retrieve.
----	--------------	---------------------------------

Returns

A const reference to the event log entry.

Definition at line 102 of file [event_manager.cpp](#).

7.11.4.5 `save_to_storage()`

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

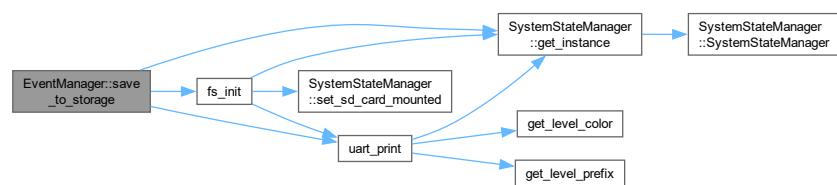
Saves the event buffer to persistent storage.

Returns

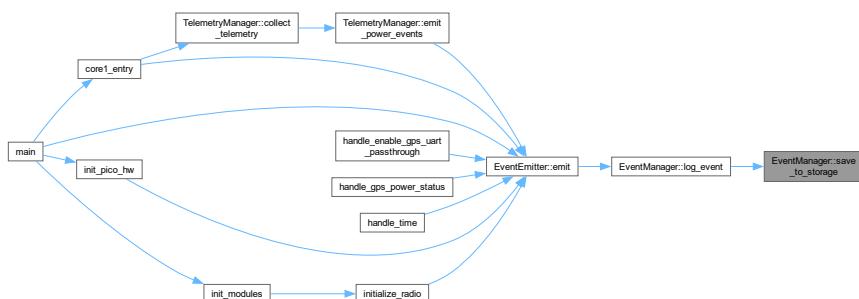
True if the save was successful, false otherwise.

Definition at line 128 of file [event_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.11.5 Variable Documentation

7.11.5.1 `__attribute__`

```
class EventManager __attribute__
```

7.12 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.

7.12.1 Detailed Description

Classes for handling location data.

7.12.2 Macro Definition Documentation

7.12.2.1 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](#).

7.12.3 Function Documentation

7.12.3.1 splitString()

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

Splits a string into tokens based on a delimiter.

Parameters

in	<i>str</i>	The string to split.
in	<i>delimiter</i>	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:

**7.12.3.2 collect_gps_data()**

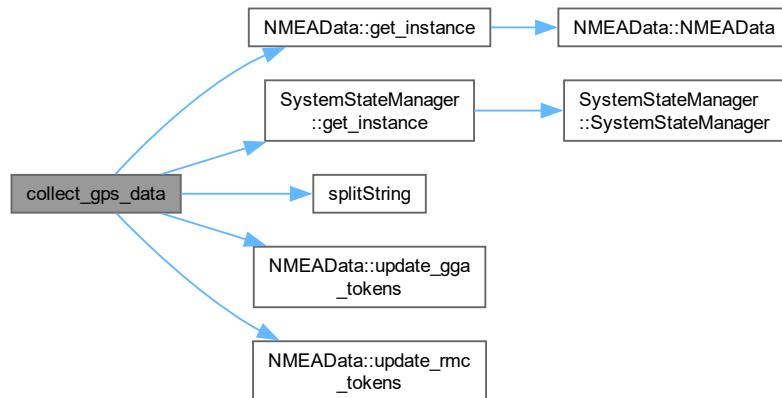
```
void collect_gps_data ()
```

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:

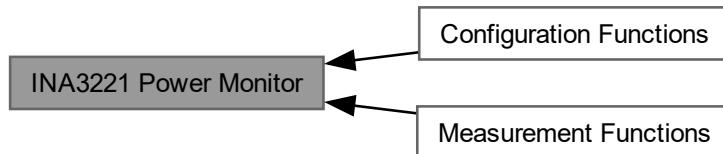


Here is the caller graph for this function:



7.13 INA3221 Power Monitor

Collaboration diagram for INA3221 Power Monitor:



Topics

- Configuration Functions
- Measurement Functions

7.13.1 Detailed Description

7.13.2 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.
- `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_continuous ()`
Set device to continuous measurement mode.
- `void INA3221::set_mode_triggered ()`
Set device to triggered measurement mode.
- `void INA3221::set_averaging_mode (ina3221_avg_mode_t mode)`
Set the averaging mode for measurements.

7.13.2.1 Detailed Description

Functions for configuring the [INA3221](#) device

7.13.2.2 Function Documentation

7.13.2.2.1 INA3221()

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

Constructor for [INA3221](#) class.

Parameters

<i>addr</i>	I2C address of the device
<i>i2c</i>	Pointer to I2C instance

Definition at line 41 of file [INA3221.cpp](#).

7.13.2.2.2 begin()

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

Initialize the [INA3221](#) device.

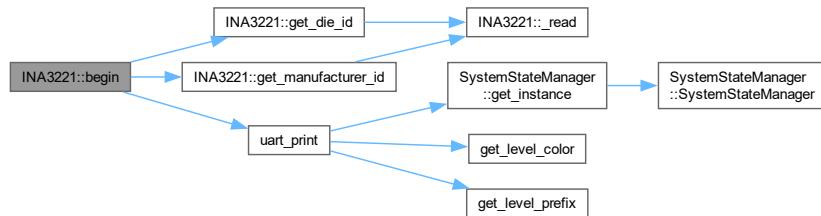
Returns

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:



7.13.2.2.3 `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 83 of file [INA3221.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.2.2.4 `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 95 of file [INA3221.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.2.2.5 `read_register()`

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

Read a register from the device.

Parameters

<i>reg</i>	Register address to read
------------	--------------------------

Returns

16-bit value read from the register

Definition at line 108 of file [INA3221.cpp](#).

Here is the call graph for this function:



7.13.2.2.6 set_mode_continuous()

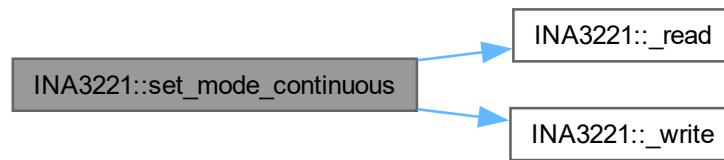
```
void INA3221::set_mode_continuous ()
```

Set device to continuous measurement mode.

Enables continuous measurement of bus voltage and shunt voltage

Definition at line 122 of file [INA3221.cpp](#).

Here is the call graph for this function:



7.13.2.2.7 set_mode_triggered()

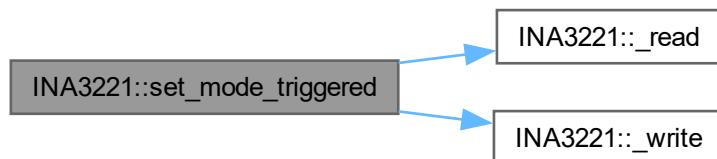
```
void INA3221::set_mode_triggered ()
```

Set device to triggered measurement mode.

Disables continuous measurements, requiring manual triggers

Definition at line 136 of file [INA3221.cpp](#).

Here is the call graph for this function:



7.13.2.2.8 set_averaging_mode()

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

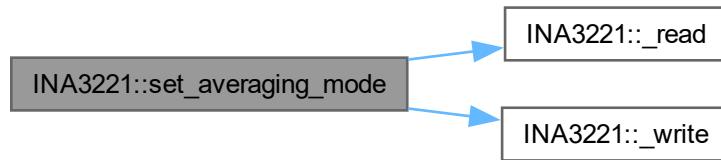
Set the averaging mode for measurements.

Parameters

<code>mode</code>	Number of samples to average
-------------------	------------------------------

Definition at line 149 of file [INA3221.cpp](#).

Here is the call graph for this function:



7.13.3 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.

7.13.3.1 Detailed Description

Functions for reading voltage, current and power measurements

7.13.3.2 Function Documentation

7.13.3.2.1 `get_shunt_voltage()`

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

Get shunt voltage for a specific channel.

Parameters

<i>channel</i>	Channel number (1-3)
----------------	----------------------

Returns

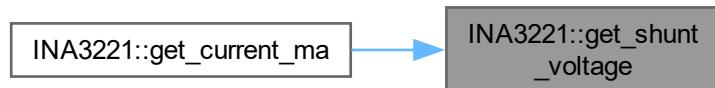
Shunt voltage in microvolts (μ V)

Definition at line 164 of file [INA3221.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.3.2.2 `get_current_ma()`

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

Get current for a specific channel.

Parameters

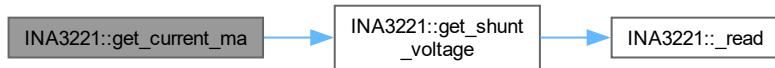
<i>channel</i>	Channel number (1-3)
----------------	----------------------

Returns

Current in millamps (mA)

Definition at line 196 of file [INA3221.cpp](#).

Here is the call graph for this function:



7.13.3.2.3 get_voltage()

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

Get bus voltage for a specific channel.

Parameters

<i>channel</i>	Channel number (1-3)
----------------	----------------------

Returns

Voltage in volts (V)

Definition at line 212 of file [INA3221.cpp](#).

Here is the call graph for this function:



7.14 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_voltage_solar ()**
Gets solar 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 (ina3221_op_mode_t op_mode, ina3221_avg_mode_t avg_mode)**
Configures the [INA3221](#).

7.14.1 Detailed Description

Classes for handling power-related functions.

7.14.2 Function Documentation

7.14.2.1 PowerManager()

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

Private constructor for the singleton pattern.

Initializes the [INA3221](#) and mutex.

Definition at line 25 of file [PowerManager.cpp](#).

7.14.2.2 `get_instance()`

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

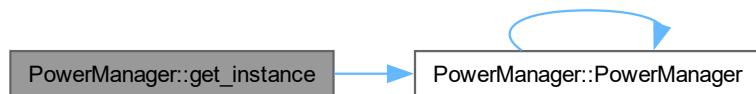
Gets the singleton instance of the [PowerManager](#) class.

Returns

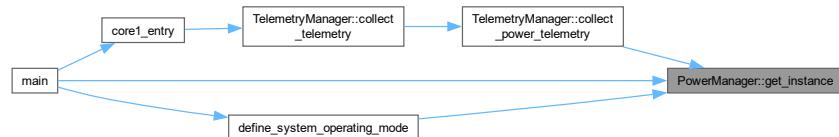
A reference to the singleton instance.

Definition at line 35 of file [PowerManager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.14.2.3 `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:



7.14.2.4 `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](#).

7.14.2.5 `get_voltage_battery()`

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

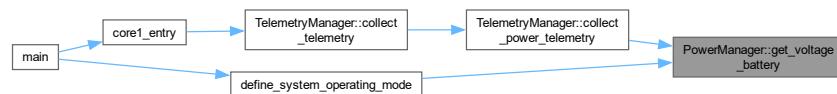
Gets the battery voltage.

Returns

The battery voltage in volts.

Definition at line [77](#) of file [PowerManager.cpp](#).

Here is the caller graph for this function:



7.14.2.6 `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:



7.14.2.7 `get_voltage_solar()`

```
float PowerManager::get_voltage_solar ()
```

Gets solar voltage.

Gets the solar voltage.

Returns

The solar voltage in volts.

Definition at line 103 of file [PowerManager.cpp](#).

7.14.2.8 `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 116 of file [PowerManager.cpp](#).

Here is the caller graph for this function:



7.14.2.9 `get_current_draw()`

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

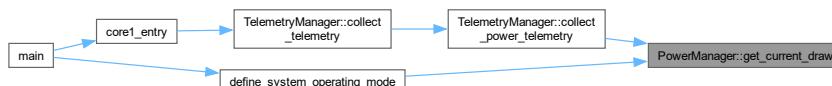
Gets the current draw.

Returns

The current draw in milliamperes.

Definition at line 129 of file [PowerManager.cpp](#).

Here is the caller graph for this function:



7.14.2.10 `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 142 of file [PowerManager.cpp](#).

Here is the caller graph for this function:



7.14.2.11 `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 155 of file [PowerManager.cpp](#).

7.14.2.12 `configure()`

```
void PowerManager::configure (
    ina3221_op_mode_t op_mode,
    ina3221_avg_mode_t avg_mode)
```

Configures the [INA3221](#).

Parameters

in	<i>config</i>	A map of configuration parameters.
in	<i>op_mode</i>	Operating mode
in	<i>avg_mode</i>	Averaging mode.

Definition at line 169 of file [PowerManager.cpp](#).

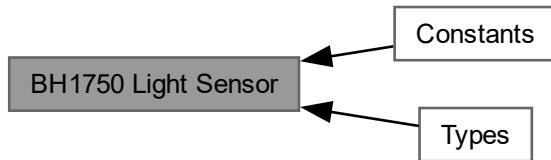
Here is the caller graph for this function:



7.15 BH1750 Light Sensor

Driver for the [BH1750](#) digital light sensor.

Collaboration diagram for BH1750 Light Sensor:



Topics

- [Constants](#)
Defines constants used by the [BH1750](#) driver.
- [Types](#)
Defines types used by the [BH1750](#) driver.

Classes

- class [BH1750](#)
Class to interface with the [BH1750](#) light sensor.

Functions

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

7.15.1 Detailed Description

Driver for the [BH1750](#) digital light sensor.

7.15.2 Function Documentation

7.15.2.1 BH1750()

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

Constructor for the [BH1750](#) class.

Parameters

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

Definition at line 20 of file [BH1750.cpp](#).

7.15.2.2 begin()

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

Initializes the [BH1750](#) sensor.

Parameters

<i>mode</i>	Measurement mode to use (default: CONTINUOUS_HIGH_RES_MODE).
-------------	--

Returns

True if initialization was successful, false otherwise.

Definition at line 28 of file [BH1750.cpp](#).

Here is the call graph for this function:



7.15.2.3 configure()

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

Configures the [BH1750](#) sensor with the specified mode.

Parameters

<i>mode</i>	Measurement mode to configure.
-------------	--------------------------------

Returns

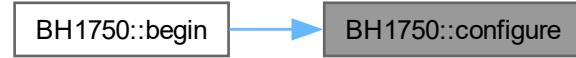
True if configuration was successful, false otherwise.

Definition at line 42 of file [BH1750.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.15.2.4 `get_light_level()`

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

Reads the light level from the [BH1750](#) sensor.

Returns

Light level in lux.

Definition at line 67 of file [BH1750.cpp](#).

7.15.2.5 `write8()`

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

Writes a single byte of data to the [BH1750](#) sensor.

Parameters

<code>data</code>	Byte of data to write.
-------------------	------------------------

Definition at line 81 of file [BH1750.cpp](#).

Here is the caller graph for this function:



7.15.3 Constants

Defines constants used by the [BH1750](#) driver.

Collaboration diagram for Constants:



Macros

- `#define _BH1750_DEVICE_ID 0xE1`
Correct content of WHO_AM_I register (not actually used in this driver).
- `#define _BH1750_MTREG_MIN 31`
Minimum value for the MTREG register.
- `#define _BH1750_MTREG_MAX 254`
Maximum value for the MTREG register.
- `#define _BH1750_DEFAULT_MTREG 69`
Default value for the MTREG register.

7.15.3.1 Detailed Description

Defines constants used by the [BH1750](#) driver.

7.15.3.2 Macro Definition Documentation

7.15.3.2.1 _BH1750_DEVICE_ID

```
#define _BH1750_DEVICE_ID 0xE1
```

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

Definition at line 36 of file [BH1750.h](#).

7.15.3.2.2 _BH1750_MTREG_MIN

```
#define _BH1750_MTREG_MIN 31
```

Minimum value for the MTREG register.

Definition at line 42 of file [BH1750.h](#).

7.15.3.2.3 _BH1750_MTREG_MAX

```
#define _BH1750_MTREG_MAX 254
```

Maximum value for the MTREG register.

Definition at line 48 of file [BH1750.h](#).

7.15.3.2.4 _BH1750_DEFAULT_MTREG

```
#define _BH1750_DEFAULT_MTREG 69
```

Default value for the MTREG register.

Definition at line 54 of file [BH1750.h](#).

7.15.4 Types

Defines types used by the [BH1750](#) driver.

Collaboration diagram for Types:



Enumerations

- enum class `BH1750::Mode` : `uint8_t` {

`BH1750::Mode::UNCONFIGURED_POWER_DOWN` = `0x00` , `BH1750::Mode::POWER_ON` = `0x01` ,

`BH1750::Mode::RESET` = `0x07` , `BH1750::Mode::CONTINUOUS_HIGH_RES_MODE` = `0x10` ,

`BH1750::Mode::CONTINUOUS_HIGH_RES_MODE_2` = `0x11` , `BH1750::Mode::CONTINUOUS_LOW_RES_MODE`

`= 0x13` , `BH1750::Mode::ONE_TIME_HIGH_RES_MODE` = `0x20` , `BH1750::Mode::ONE_TIME_HIGH_RES_MODE_2`

`= 0x21` ,

`BH1750::Mode::ONE_TIME_LOW_RES_MODE` = `0x23` }

Enumeration of measurement modes for the `BH1750` sensor.

7.15.4.1 Detailed Description

Defines types used by the `BH1750` driver.

7.15.4.2 Enumeration Type Documentation

7.15.4.2.1 Mode

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

Enumeration of measurement modes for the `BH1750` sensor.

Enumerator

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

Definition at line 66 of file `BH1750.h`.

7.16 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 `SensorDataTypelIdentifier` : `uint8_t` {
 `SensorDataTypelIdentifier::NONE` = `0x00` , `SensorDataTypelIdentifier::LIGHT_LEVEL` = `0x01` , `SensorDataTypelIdentifier::TEMPERATURE` = `0x02` , `SensorDataTypelIdentifier::HUMIDITY` = `0x03` , `SensorDataTypelIdentifier::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, `SensorDataTypelIdentifier` dataType)

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

Gets a sensor.

7.16.1 Detailed Description

Classes for handling sensor-related functions.

7.16.2 Enumeration Type Documentation

7.16.2.1 SensorType

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

Enumeration of sensor types.

Defines the different types of sensors that can be managed.

Enumerator

<code>NONE</code>	No sensor.
<code>LIGHT</code>	Light sensor.
<code>ENVIRONMENT</code>	Environment sensor.

Definition at line 31 of file `ISensor.h`.

7.16.2.2 SensorDataTypelIdentifier

```
enum class SensorDataTypelIdentifier : 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](#).

7.16.3 Function Documentation

7.16.3.1 sensor_init()

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

Initializes a sensor.

Parameters

in	<i>type</i>	Sensor type to initialize.
in	<i>i2c</i>	I2C instance to use for communication.

Returns

True if initialization was successful, false otherwise.

Definition at line 27 of file [ISensor.cpp](#).

Here is the caller graph for this function:



7.16.3.2 sensor_configure()

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

Configures a sensor.

Parameters

in	<i>type</i>	Sensor type to configure.
in	<i>config</i>	A map of configuration parameters.

Returns

True if configuration was successful, false otherwise.

Definition at line 48 of file [ISensor.cpp](#).

7.16.3.3 sensor_read_data()

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

Reads data from a sensor.

Parameters

in	<i>sensorType</i>	Sensor type to read from.
in	<i>dataType</i>	Data type to read.

Returns

The sensor data.

Definition at line 62 of file [ISensor.cpp](#).

Here is the caller graph for this function:

**7.16.3.4 get_sensor()**

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

Gets a sensor.

Parameters

in	<i>type</i>	Sensor type to get.
----	-------------	---------------------

Returns

A pointer to the sensor.

Definition at line 75 of file [ISensor.cpp](#).

7.17 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.

7.17.1 Detailed Description

Classes and functions for managing file system operations.

7.17.2 Function Documentation

7.17.2.1 **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.

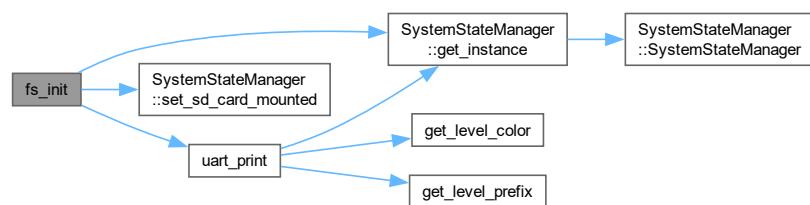
Returns

True if initialization was successful, false otherwise.

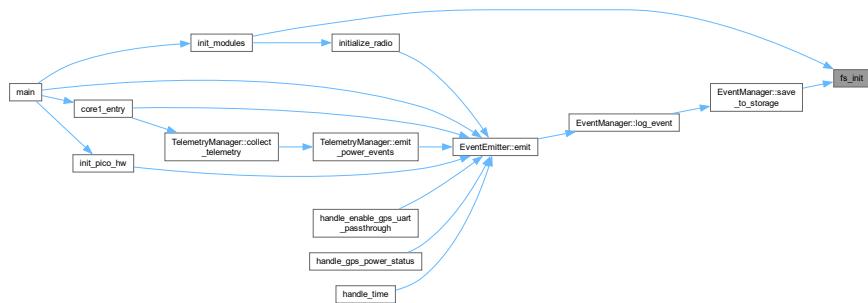
Mounts the FAT file system on the SD card. If mounting fails, it formats the SD card with FAT and then attempts to mount again.

Definition at line 25 of file [storage.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.17.2.2 fs_stop()

```
bool fs_stop (
    void )
```

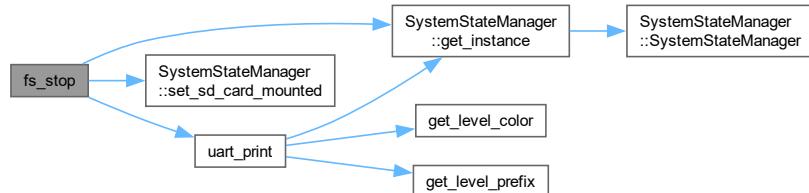
Unmounts the file system from the SD card.

Returns

True if unmounting was successful, false otherwise.

Definition at line 65 of file [storage.cpp](#).

Here is the call graph for this function:



7.18 System State Manager

Classes for handling system state management.

Classes

- class [SystemStateManager](#)

Manages the system state of the Kubisat firmware.

Enumerations

- enum class `SystemOperatingMode` : `uint8_t` { `SystemOperatingMode::BATTERY_POWERED` = 0 ,
`SystemOperatingMode::USB_POWERED` = 1 }
Enumeration of system operating modes.

7.18.1 Detailed Description

Classes for handling system state management.

7.18.2 Enumeration Type Documentation

7.18.2.1 SystemOperatingMode

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

Enumeration of system operating modes.

Enumerator

<code>BATTERY_POWERED</code>	Battery powered mode
<code>USB_POWERED</code>	USB powered mode

Definition at line 27 of file `system_state_manager.h`.

7.19 Telemetry Manager

Classes

- struct `TelemetryRecord`
Structure representing a single telemetry data point.
- struct `SensorDataRecord`
Structure representing a single sensor data point.
- class `TelemetryManager`
Manages the collection, storage, and retrieval of telemetry data.

Macros

- #define `TELEMETRY_CSV_PATH` "/telemetry.csv"
Path to the telemetry CSV file on storage media.
- #define `SENSOR_DATA_CSV_PATH` "/sensors.csv"
Path to the sensor data CSV file on storage media.
- #define `DEFAULT_SAMPLE_INTERVAL_MS` 1000
Default interval between telemetry samples in milliseconds (2 seconds)
- #define `DEFAULT_FLUSH_THRESHOLD` 10
Default number of records to collect before flushing to storage.

Functions

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

7.19.1 Detailed Description

7.19.2 Macro Definition Documentation

7.19.2.1 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](#).

7.19.2.2 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](#).

7.19.2.3 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](#).

7.19.2.4 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](#).

7.19.3 Function Documentation

7.19.3.1 to_csv() [1/2]

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

Converts the telemetry record to a CSV string.

Returns

A CSV string representing the telemetry record.

Definition at line [77](#) of file [telemetry_manager.h](#).

7.19.3.2 to_csv() [2/2]

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

Converts the sensor data record to a CSV string.

Returns

A CSV string representing the sensor data record.

Definition at line [123](#) of file [telemetry_manager.h](#).

7.19.3.3 collect_power_telemetry()

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

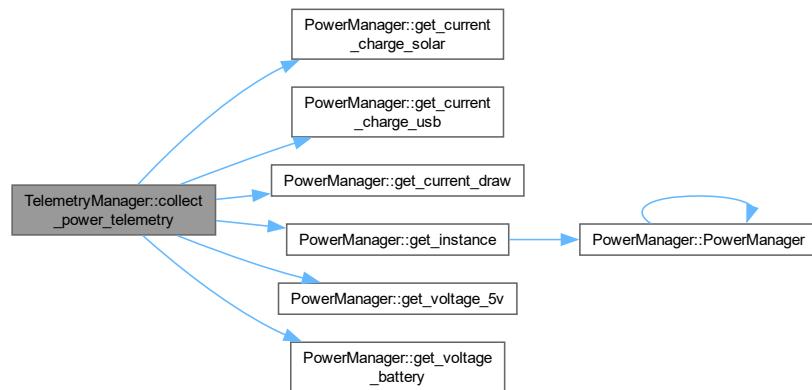
Collects power subsystem telemetry data.

Parameters

<code>out</code>	<code>record</code>	The telemetry record to update with power data.
------------------	---------------------	---

Definition at line 108 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.19.3.4 `emit_power_events()`

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

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

Parameters

<code>in</code>	<code>battery_voltage</code>	The current battery voltage.
<code>in</code>	<code>charge_current_usb</code>	The current USB charging current.
<code>in</code>	<code>charge_current_solar</code>	The current solar charging current.
<code>in</code>	<code>discharge_current</code>	The current battery discharge current.

Definition at line 124 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.19.3.5 collect_gps_telemetry()

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

Collects GPS telemetry data.

Parameters

<code>out</code>	<code>record</code>	The telemetry record to update with GPS data.
------------------	---------------------	---

Definition at line 183 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.19.3.6 collect_sensor_telemetry()

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

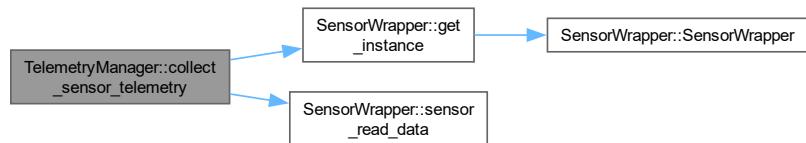
Collects sensor telemetry data.

Parameters

<code>out</code>	<code>sensor_record</code>	The sensor data record to update with sensor data.
------------------	----------------------------	--

Definition at line 233 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

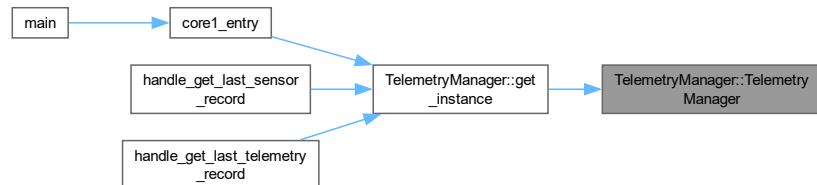


7.19.3.7 TelemetryManager()

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

Definition at line 44 of file [telemetry_manager.cpp](#).

Here is the caller graph for this function:



7.19.3.8 init()

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

Initialize the telemetry system.

Initializes the telemetry manager.

Returns

True if initialization was successful

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

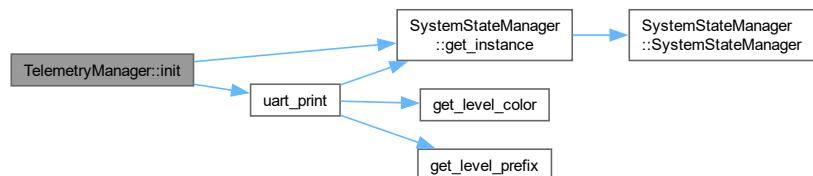
Returns

True if initialization was successful, false otherwise.

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

Definition at line 54 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.19.3.9 collect_telemetry()

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

Collect telemetry data from sensors and power subsystems.

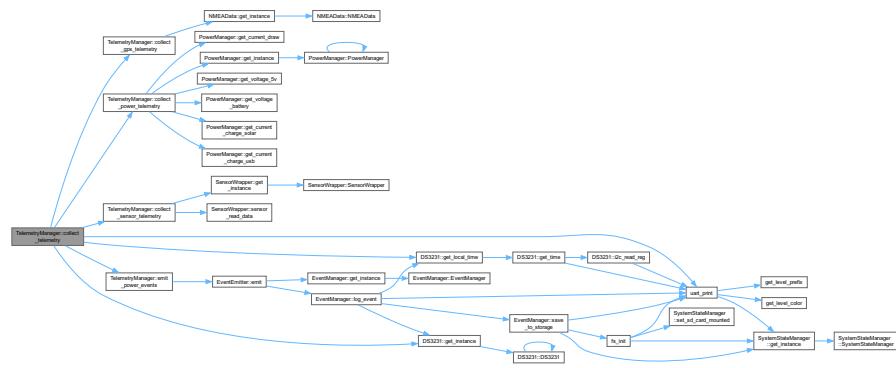
Returns

True if data was successfully collected

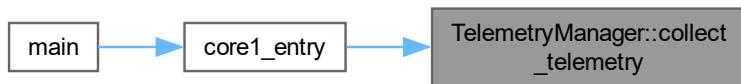
Reads data from power manager, sensors, and GPS and stores it in the telemetry buffer with proper mutex protection

Definition at line 248 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.19.3.10 flush_telemetry()

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

Save buffered telemetry data to storage.

Save buffered telemetry and sensor data to storage.

Returns

True if data was successfully saved

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

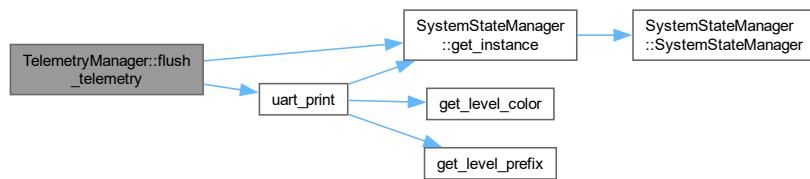
Returns

True if data was successfully saved

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

Definition at line 290 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.19.3.11 is_telemetry_collection_time()**

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

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

Parameters

<i>current_time</i>	Current system time in milliseconds
<i>last_collection_time</i>	Previous collection time in milliseconds

Returns

True if collection interval has passed

Updates *last_collection_time* if the interval has passed

Definition at line 346 of file [telemetry_manager.cpp](#).

7.19.3.12 is_telemetry_flush_time()

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

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

Parameters

<i>collection_counter</i>	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 [362](#) of file [telemetry_manager.cpp](#).

7.19.3.13 get_last_telemetry_record_csv()

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

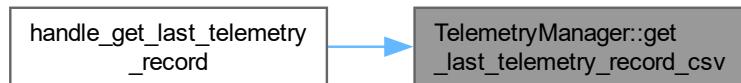
Gets the last telemetry record as a CSV string.

Returns

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

Definition at line [375](#) of file [telemetry_manager.cpp](#).

Here is the caller graph for this function:



7.19.3.14 `get_last_sensor_record_csv()`

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

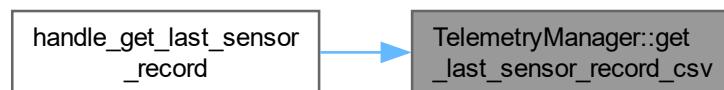
Gets the last sensor data record as a CSV string.

Returns

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

Definition at line 384 of file [telemetry_manager.cpp](#).

Here is the caller graph for this function:



Chapter 8

Class Documentation

8.1 BH1750 Class Reference

Class to interface with the [BH1750](#) light sensor.

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

Public Types

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

Enumeration of measurement modes for the [BH1750](#) sensor.

Public Member Functions

- `BH1750 (i2c_inst_t *i2c, uint8_t addr=0x23)`
Constructor for the [BH1750](#) class.
- `bool begin (Mode mode=Mode::CONTINUOUS_HIGH_RES_MODE)`
Initializes the [BH1750](#) sensor.
- `bool configure (Mode mode)`
Configures the [BH1750](#) sensor with the specified mode.
- `float get_light_level ()`
Reads the light level from the [BH1750](#) sensor.

Private Member Functions

- `void write8 (uint8_t data)`
Writes a single byte of data to the [BH1750](#) sensor.

Private Attributes

- `uint8_t _i2c_addr`
I2C address of the [BH1750](#) sensor.
- `i2c_inst_t * i2c_port_`
Pointer to the I2C interface.

8.1.1 Detailed Description

Class to interface with the [BH1750](#) light sensor.

Definition at line [60](#) of file [BH1750.h](#).

8.1.2 Member Data Documentation

8.1.2.1 `_i2c_addr`

`uint8_t BH1750::_i2c_addr [private]`

I2C address of the [BH1750](#) sensor.

Definition at line [122](#) of file [BH1750.h](#).

8.1.2.2 `i2c_port_`

`i2c_inst_t* BH1750::i2c_port_ [private]`

Pointer to the I2C interface.

Definition at line [124](#) of file [BH1750.h](#).

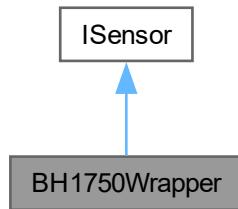
The documentation for this class was generated from the following files:

- lib/sensors/BH1750/[BH1750.h](#)
- lib/sensors/BH1750/[BH1750.cpp](#)

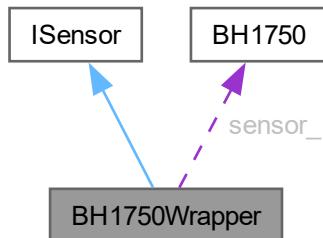
8.2 BH1750Wrapper Class Reference

```
#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 \(SensorDataTypedef identifier type\) 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`

8.2.1 Detailed Description

Definition at line 9 of file [BH1750_WRAPPER.h](#).

8.2.2 Constructor & Destructor Documentation

8.2.2.1 `BH1750Wrapper()` [1/2]

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

Definition at line 5 of file [BH1750_WRAPPER.cpp](#).

8.2.2.2 `BH1750Wrapper()` [2/2]

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

8.2.3 Member Function Documentation

8.2.3.1 `get_i2c_addr()`

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

8.2.3.2 `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](#).

8.2.3.3 `read_data()`

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

Reads data from the sensor.

Returns

in	<i>type</i>	Data type to read.
----	-------------	--------------------

Returns

The sensor data.

Implements [ISensor](#).

Definition at line 14 of file [BH1750_WRAPPER.cpp](#).

8.2.3.4 `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](#).

8.2.3.5 `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](#).

8.2.3.6 `configure()`

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

Configures the sensor.

Parameters

in	<i>config</i>	A map of configuration parameters.
----	---------------	------------------------------------

Returns

True if configuration was successful, false otherwise.

Implements [ISensor](#).

Definition at line [29](#) of file [BH1750_WRAPPER.cpp](#).

8.2.3.7 `get_address()`

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

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

Implements [ISensor](#).

Definition at line [25](#) of file [BH1750_WRAPPER.h](#).

8.2.4 Member Data Documentation

8.2.4.1 `sensor_`

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

Definition at line [11](#) of file [BH1750_WRAPPER.h](#).

8.2.4.2 `initialized_`

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

Definition at line [12](#) of file [BH1750_WRAPPER.h](#).

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

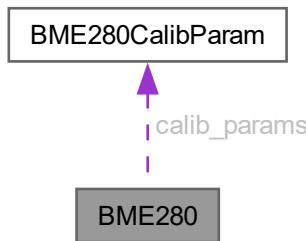
- lib/sensors/BH1750/[BH1750_WRAPPER.h](#)
- lib/sensors/BH1750/[BH1750_WRAPPER.cpp](#)

8.3 BME280 Class Reference

Class to interface with the [BME280](#) environmental sensor.

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

Collaboration diagram for BME280:



Public Types

- enum { [ADDR_SDO_LOW](#) = 0x76 , [ADDR_SDO_HIGH](#) = 0x77 }

I2C Address Options for the [BME280](#) sensor.

- enum class [Oversampling](#) : uint8_t {
 [OSR_X0](#) = 0x00 , [OSR_X1](#) = 0x01 , [OSR_X2](#) = 0x02 , [OSR_X4](#) = 0x03 ,
 [OSR_X8](#) = 0x04 , [OSR_X16](#) = 0x05 }

Enum class for oversampling settings.

Public Member Functions

- [BME280](#) (*i2c_inst_t* **i2cPort*, *uint8_t* *address*=[ADDR_SDO_LOW](#))
Constructor for the [BME280](#) class.
- bool [init](#) ()
Initializes the [BME280](#) sensor.
- void [reset](#) ()
Resets the [BME280](#) sensor.
- bool [read_raw_all](#) (*int32_t* **temperature*, *int32_t* **pressure*, *int32_t* **humidity*)
Reads all raw data from the sensor.
- float [convert_temperature](#) (*int32_t* *temp_raw*) const
Converts raw temperature data to degrees Celsius.
- float [convert_pressure](#) (*int32_t* *pressure_raw*) const
Converts raw pressure data to hectopascals (hPa).
- float [convert_humidity](#) (*int32_t* *humidity_raw*) const
Converts raw humidity data to relative humidity (%).

Private Types

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

Register Definitions for the BME280 sensor.

- enum { HUMIDITY_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16) , TEMPERATURE_OVERSAMPLING
 = static_cast<uint8_t>(Oversampling::OSR_X16) , PRESSURE_OVERSAMPLING = static_cast<uint8_t>
 (Oversampling::OSR_X16) , NORMAL_MODE = 0xB7 }

Sensor settings.

- enum { NUM_CALIB_PARAMS = 26 , NUM_HUM_CALIB_PARAMS = 7 }

Calibration data length.

Private Member Functions

- bool `write_register` (uint8_t reg, uint8_t value)
Helper function for I2C writes.
- bool `read_register` (uint8_t reg, uint8_t *data)
Helper function for I2C reads.
- bool `read_register` (uint8_t reg, uint8_t *data, size_t len)
Helper function for I2C reads with a specified length.
- bool `configure_sensor` ()
Configures the sensor with default settings.
- bool `get_calibration_parameters` ()
Retrieves the calibration parameters from the sensor.

Private Attributes

- i2c_inst_t * `i2c_port`
Pointer to the I2C interface.
- uint8_t `device_addr`
I2C device address.
- BME280CalibParam `calib_params`
Calibration parameters for the sensor.
- bool `initialized_`
Initialization status of the sensor.
- int32_t `t_fine`
Fine temperature parameter needed for compensation.

8.3.1 Detailed Description

Class to interface with the [BME280](#) environmental sensor.

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

Definition at line [71](#) of file [BME280.h](#).

8.3.2 Member Enumeration Documentation

8.3.2.1 anonymous enum

```
anonymous enum
```

I2C Address Options for the [BME280](#) sensor.

Enumerator

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

Definition at line [76](#) of file [BME280.h](#).

8.3.2.2 Oversampling

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

Enum class for oversampling settings.

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

Enumerator

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

Definition at line [89](#) of file [BME280.h](#).

8.3.2.3 anonymous enum

```
anonymous enum [private]
```

Register Definitions for the [BME280](#) sensor.

Enumerator

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

Definition at line [207](#) of file [BME280.h](#).

8.3.2.4 anonymous enum

```
anonymous enum [private]
```

Sensor settings.

Enumerator

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

Definition at line 256 of file [BME280.h](#).

8.3.2.5 anonymous enum

anonymous enum [private]

Calibration data length.

Enumerator

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

Definition at line 266 of file [BME280.h](#).

8.3.3 Constructor & Destructor Documentation

8.3.3.1 BME280()

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

Constructor for the [BME280](#) class.

Parameters

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

Definition at line 24 of file [BME280.cpp](#).

8.3.4 Member Function Documentation

8.3.4.1 init()

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

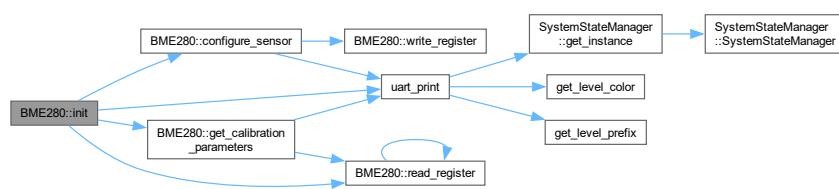
Initializes the [BME280](#) sensor.

Returns

True if initialization was successful, false otherwise.

Definition at line [32](#) of file [BME280.cpp](#).

Here is the call graph for this function:



8.3.4.2 reset()

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

Resets the [BME280](#) sensor.

Definition at line [70](#) of file [BME280.cpp](#).

Here is the call graph for this function:



8.3.4.3 read_raw_all()

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

Reads all raw data from the sensor.

Parameters

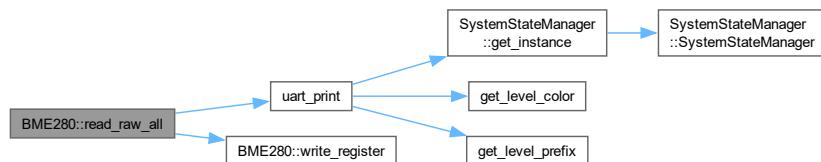
<i>temperature</i>	Pointer to store the raw temperature value.
<i>pressure</i>	Pointer to store the raw pressure value.
<i>humidity</i>	Pointer to store the raw humidity value.

Returns

True if the data was read successfully, false otherwise.

Definition at line 82 of file [BME280.cpp](#).

Here is the call graph for this function:

**8.3.4.4 convert_temperature()**

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

Converts raw temperature data to degrees Celsius.

Parameters

<i>temp_raw</i>	Raw temperature value.
-----------------	------------------------

Returns

Temperature in degrees Celsius.

Definition at line 119 of file [BME280.cpp](#).

8.3.4.5 convert_pressure()

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

Converts raw pressure data to hectopascals (hPa).

Parameters

<i>pressure_raw</i>	Raw pressure value.
---------------------	---------------------

Returns

Pressure in hPa.

Definition at line 133 of file [BME280.cpp](#).

8.3.4.6 convert_humidity()

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

Converts raw humidity data to relative humidity (%).

Parameters

<i>humidity_raw</i>	Raw humidity value.
---------------------	---------------------

Returns

Relative humidity in %.

Definition at line 159 of file [BME280.cpp](#).

8.3.4.7 write_register()

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

Helper function for I2C writes.

Parameters

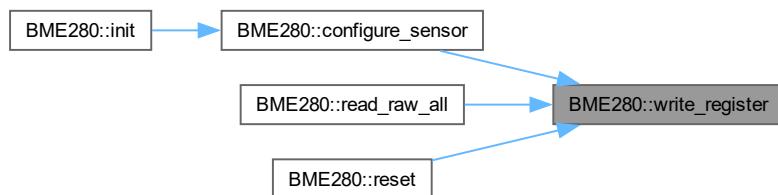
<i>reg</i>	Register address to write to.
<i>value</i>	Value to write to the register.

Returns

True if the write was successful, false otherwise.

Definition at line 250 of file [BME280.cpp](#).

Here is the caller graph for this function:



8.3.4.8 `read_register()` [1/2]

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

Helper function for I2C reads.

Parameters

<i>reg</i>	Register address to read from.
<i>data</i>	Pointer to store the read data.

Returns

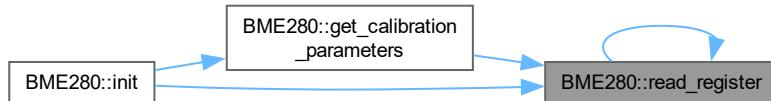
True if the read was successful, false otherwise.

Definition at line 278 of file [BME280.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.3.4.9 `read_register()` [2/2]

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

Helper function for I2C reads with a specified length.

Parameters

<i>reg</i>	Register address to read from.
<i>data</i>	Pointer to store the read data.
<i>len</i>	Number of bytes to read.

Returns

True if the read was successful, false otherwise.

Definition at line 263 of file [BME280.cpp](#).

8.3.4.10 configure_sensor()

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

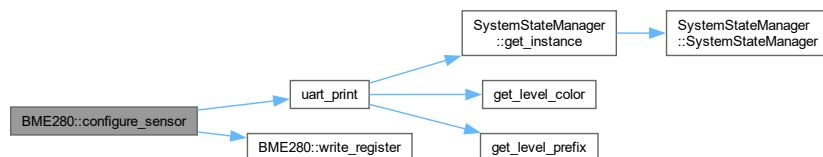
Configures the sensor with default settings.

Returns

True if the configuration was successful, false otherwise.

Definition at line 222 of file [BME280.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.3.4.11 `get_calibration_parameters()`

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

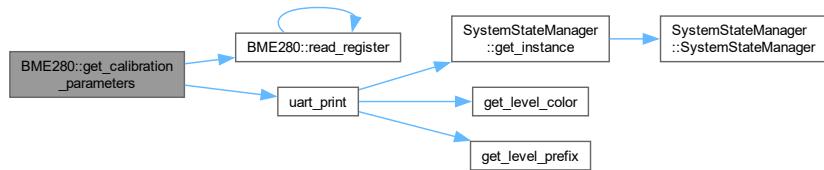
Retrieves the calibration parameters from the sensor.

Returns

True if the parameters were read successfully, false otherwise.

Definition at line 175 of file [BME280.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.3.5 Member Data Documentation

8.3.5.1 `i2c_port`

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

Pointer to the I2C interface.

Definition at line 191 of file [BME280.h](#).

8.3.5.2 `device_addr`

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

I2C device address.

Definition at line 193 of file [BME280.h](#).

8.3.5.3 calib_params

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

Calibration parameters for the sensor.

Definition at line 196 of file [BME280.h](#).

8.3.5.4 initialized_

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

Initialization status of the sensor.

Definition at line 199 of file [BME280.h](#).

8.3.5.5 t_fine

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

Fine temperature parameter needed for compensation.

Definition at line 202 of file [BME280.h](#).

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

- lib/sensors/BME280/[BME280.h](#)
- lib/sensors/BME280/[BME280.cpp](#)

8.4 BME280CalibParam Struct Reference

Structure to hold the [BME280](#) calibration parameters.

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

Public Attributes

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

8.4.1 Detailed Description

Structure to hold the [BME280](#) calibration parameters.

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

Definition at line [23](#) of file [BME280.h](#).

8.4.2 Member Data Documentation

8.4.2.1 dig_t1

```
uint16_t BME280CalibParam::dig_t1
```

Temperature calibration parameter 1.

Definition at line [25](#) of file [BME280.h](#).

8.4.2.2 dig_t2

```
int16_t BME280CalibParam::dig_t2
```

Temperature calibration parameter 2.

Definition at line [27](#) of file [BME280.h](#).

8.4.2.3 dig_t3

```
int16_t BME280CalibParam::dig_t3
```

Temperature calibration parameter 3.

Definition at line [29](#) of file [BME280.h](#).

8.4.2.4 dig_p1

```
uint16_t BME280CalibParam::dig_p1
```

Pressure calibration parameter 1.

Definition at line [32](#) of file [BME280.h](#).

8.4.2.5 dig_p2

```
int16_t BME280CalibParam::dig_p2
```

Pressure calibration parameter 2.

Definition at line [34](#) of file [BME280.h](#).

8.4.2.6 dig_p3

```
int16_t BME280CalibParam::dig_p3
```

Pressure calibration parameter 3.

Definition at line [36](#) of file [BME280.h](#).

8.4.2.7 `dig_p4`

```
int16_t BME280CalibParam::dig_p4
```

Pressure calibration parameter 4.

Definition at line [38](#) of file [BME280.h](#).

8.4.2.8 `dig_p5`

```
int16_t BME280CalibParam::dig_p5
```

Pressure calibration parameter 5.

Definition at line [40](#) of file [BME280.h](#).

8.4.2.9 `dig_p6`

```
int16_t BME280CalibParam::dig_p6
```

Pressure calibration parameter 6.

Definition at line [42](#) of file [BME280.h](#).

8.4.2.10 `dig_p7`

```
int16_t BME280CalibParam::dig_p7
```

Pressure calibration parameter 7.

Definition at line [44](#) of file [BME280.h](#).

8.4.2.11 `dig_p8`

```
int16_t BME280CalibParam::dig_p8
```

Pressure calibration parameter 8.

Definition at line [46](#) of file [BME280.h](#).

8.4.2.12 `dig_p9`

```
int16_t BME280CalibParam::dig_p9
```

Pressure calibration parameter 9.

Definition at line [48](#) of file [BME280.h](#).

8.4.2.13 dig_h1

```
uint8_t BME280CalibParam::dig_h1
```

Humidity calibration parameter 1.

Definition at line 51 of file [BME280.h](#).

8.4.2.14 dig_h2

```
int16_t BME280CalibParam::dig_h2
```

Humidity calibration parameter 2.

Definition at line 53 of file [BME280.h](#).

8.4.2.15 dig_h3

```
uint8_t BME280CalibParam::dig_h3
```

Humidity calibration parameter 3.

Definition at line 55 of file [BME280.h](#).

8.4.2.16 dig_h4

```
int16_t BME280CalibParam::dig_h4
```

Humidity calibration parameter 4.

Definition at line 57 of file [BME280.h](#).

8.4.2.17 dig_h5

```
int16_t BME280CalibParam::dig_h5
```

Humidity calibration parameter 5.

Definition at line 59 of file [BME280.h](#).

8.4.2.18 dig_h6

```
int8_t BME280CalibParam::dig_h6
```

Humidity calibration parameter 6.

Definition at line 61 of file [BME280.h](#).

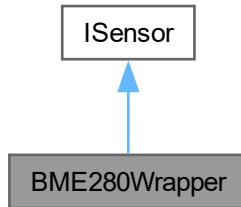
The documentation for this struct was generated from the following file:

- lib/sensors/BME280/[BME280.h](#)

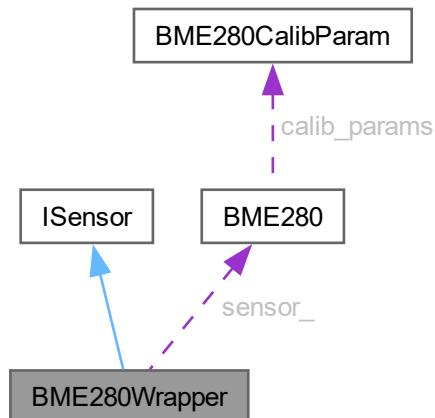
8.5 BME280Wrapper Class Reference

```
#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 (SensorDataTypedef Identifier 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

8.5.1 Detailed Description

Definition at line 8 of file [BME280_WRAPPER.h](#).

8.5.2 Constructor & Destructor Documentation

8.5.2.1 `BME280Wrapper()`

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

Definition at line 3 of file [BME280_WRAPPER.cpp](#).

8.5.3 Member Function Documentation

8.5.3.1 `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](#).

8.5.3.2 `read_data()`

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

Reads data from the sensor.

Parameters

in	<i>type</i>	Data type to read.
----	-------------	--------------------

Returns

The sensor data.

Implements [ISensor](#).

Definition at line 10 of file [BME280_WRAPPER.cpp](#).

8.5.3.3 `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](#).

8.5.3.4 `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](#).

8.5.3.5 `configure()`

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

Configures the sensor.

Parameters

in	<i>config</i>	A map of configuration parameters.
----	---------------	------------------------------------

Returns

True if configuration was successful, false otherwise.

Implements [ISensor](#).

Definition at line 34 of file [BME280_WRAPPER.cpp](#).

8.5.3.6 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](#).

8.5.4 Member Data Documentation**8.5.4.1 sensor_**

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

Definition at line 10 of file [BME280_WRAPPER.h](#).

8.5.4.2 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](#)

8.6 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`

8.6.1 Detailed Description

Configuration register bit fields.

Definition at line 91 of file [INA3221.h](#).

8.6.2 Member Data Documentation

8.6.2.1 mode_shunt_en

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

Definition at line 92 of file [INA3221.h](#).

8.6.2.2 mode_bus_en

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

Definition at line 93 of file [INA3221.h](#).

8.6.2.3 mode_continious_en

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

Definition at line 94 of file [INA3221.h](#).

8.6.2.4 shunt_conv_time

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

Definition at line 95 of file [INA3221.h](#).

8.6.2.5 bus_conv_time

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

Definition at line 96 of file [INA3221.h](#).

8.6.2.6 avg_mode

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

Definition at line 97 of file [INA3221.h](#).

8.6.2.7 ch3_en

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

Definition at line 98 of file [INA3221.h](#).

8.6.2.8 ch2_en

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

Definition at line 99 of file [INA3221.h](#).

8.6.2.9 ch1_en

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

Definition at line 100 of file [INA3221.h](#).

8.6.2.10 reset

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

Definition at line 101 of file [INA3221.h](#).

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

- lib/powerman/INA3221/[INA3221.h](#)

8.7 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 (time_t unix_time)`
Sets the RTC time using a Unix timestamp.
- `time_t get_time ()`
Gets the current RTC time as Unix timestamp.
- `int read_temperature (float *resolution)`
Reads the current temperature from the [DS3231](#).
- `int16_t get_timezone_offset () const`
Gets the current timezone offset.
- `void set_timezone_offset (int16_t offset_minutes)`
Sets the timezone offset.
- `time_t get_local_time ()`
Gets the current local time (including timezone offset)

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](#).

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`

8.7.1 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](#).

8.7.2 Constructor & Destructor Documentation

8.7.2.1 DS3231() [1/2]

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

Constructor for the [DS3231](#) class.

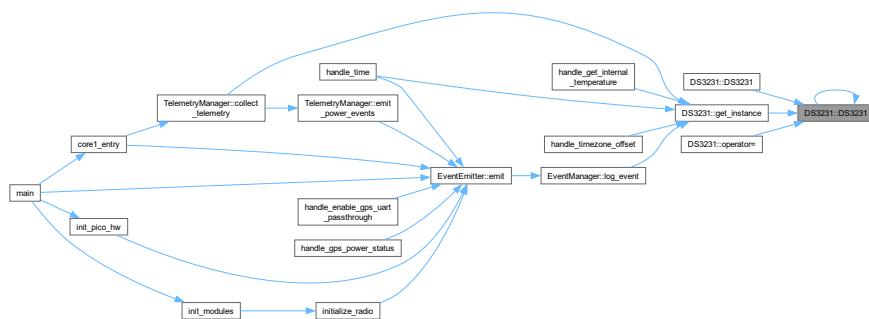
Parameters

in	<i>i2c_instance</i>	Pointer to the I2C instance to use
----	---------------------	------------------------------------

Here is the call graph for this function:



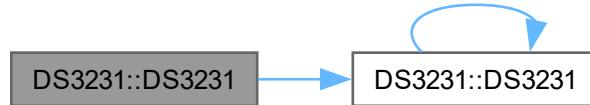
Here is the caller graph for this function:



8.7.2.2 DS3231() [2/2]

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

Here is the call graph for this function:



8.7.3 Member Function Documentation

8.7.3.1 operator=(const DS3231&)

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

Here is the call graph for this function:



8.7.4 Member Data Documentation

8.7.4.1 i2c

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

Definition at line 169 of file [DS3231.h](#).

8.7.4.2 ds3231_addr

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

Definition at line 170 of file [DS3231.h](#).

8.7.4.3 `clock_mutex_`

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

Definition at line 171 of file [DS3231.h](#).

8.7.4.4 `timezone_offset_minutes_`

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

Definition at line 172 of file [DS3231.h](#).

8.7.4.5 `sync_interval_minutes_`

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

Definition at line 173 of file [DS3231.h](#).

8.7.4.6 `last_sync_time_`

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

Definition at line 174 of file [DS3231.h](#).

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

- lib/clock/[DS3231.h](#)
- lib/clock/[DS3231.cpp](#)

8.8 `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)

8.8.1 Detailed Description

Structure to hold time and date information from [DS3231](#).

Definition at line 90 of file [DS3231.h](#).

8.8.2 Member Data Documentation

8.8.2.1 seconds

```
uint8_t ds3231_data_t::seconds
```

Seconds (0-59)

Definition at line 91 of file [DS3231.h](#).

8.8.2.2 minutes

```
uint8_t ds3231_data_t::minutes
```

Minutes (0-59)

Definition at line 92 of file [DS3231.h](#).

8.8.2.3 hours

```
uint8_t ds3231_data_t::hours
```

Hours (0-23)

Definition at line 93 of file [DS3231.h](#).

8.8.2.4 day

```
uint8_t ds3231_data_t::day
```

Day of the week (1-7)

Definition at line 94 of file [DS3231.h](#).

8.8.2.5 date

```
uint8_t ds3231_data_t::date
```

Date (1-31)

Definition at line 95 of file [DS3231.h](#).

8.8.2.6 month

`uint8_t ds3231_data_t::month`

Month (1-12)

Definition at line 96 of file [DS3231.h](#).

8.8.2.7 year

`uint8_t ds3231_data_t::year`

Year (0-99)

Definition at line 97 of file [DS3231.h](#).

8.8.2.8 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](#)

8.9 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.

8.9.1 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 264 of file [event_manager.h](#).

8.9.2 Member Function Documentation

8.9.2.1 emit()

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

Emits an event.

Parameters

in	<i>group</i>	Event group.
in	<i>event</i>	Event code.

Template Parameters

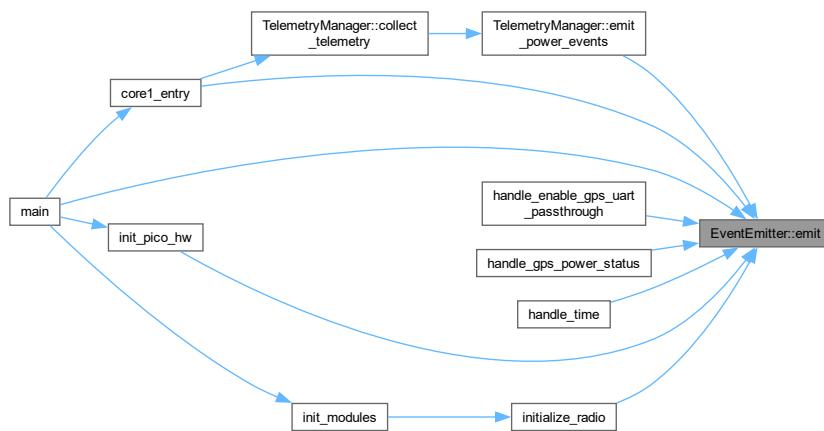
<i>T</i>	Type of the event enumeration.
----------	--------------------------------

Definition at line 273 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](#)

8.10 EventLog Class Reference

Structure for storing event log data.

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

Public Attributes

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

8.10.1 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 173 of file [event_manager.h](#).

8.10.2 Member Data Documentation

8.10.2.1 timestamp

```
uint32_t EventLog::timestamp
```

Timestamp of the event in milliseconds since boot.

Definition at line 176 of file [event_manager.h](#).

8.10.2.2 id

```
uint16_t EventLog::id
```

Unique identifier for the event.

Definition at line 178 of file [event_manager.h](#).

8.10.2.3 group

```
uint8_t EventLog::group
```

Event group.

Definition at line 180 of file [event_manager.h](#).

8.10.2.4 event

```
uint8_t EventLog::event
```

Event code.

Definition at line 182 of file [event_manager.h](#).

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

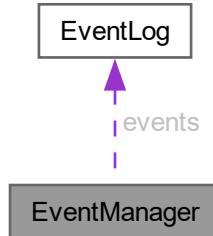
- lib/eventman/[event_manager.h](#)

8.11 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.

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`

8.11.1 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 194 of file `event_manager.h`.

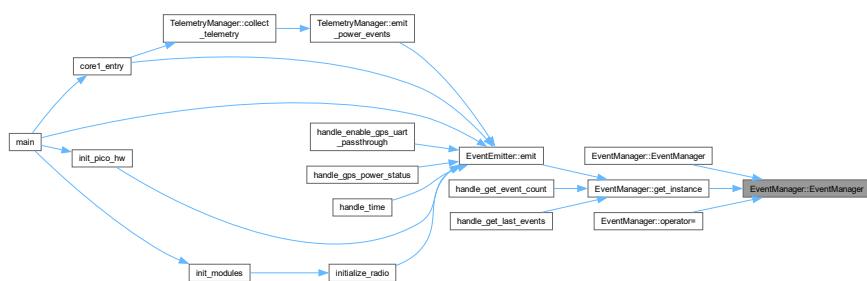
8.11.2 Constructor & Destructor Documentation

8.11.2.1 `EventManager()` [1/2]

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

Definition at line 203 of file `event_manager.h`.

Here is the caller graph for this function:



8.11.2.2 EventManager() [2/2]

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

Here is the call graph for this function:



8.11.3 Member Function Documentation

8.11.3.1 operator=()

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

Here is the call graph for this function:



8.11.3.2 get_instance()

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

Gets the singleton instance of the [EventManager](#) class.

Returns

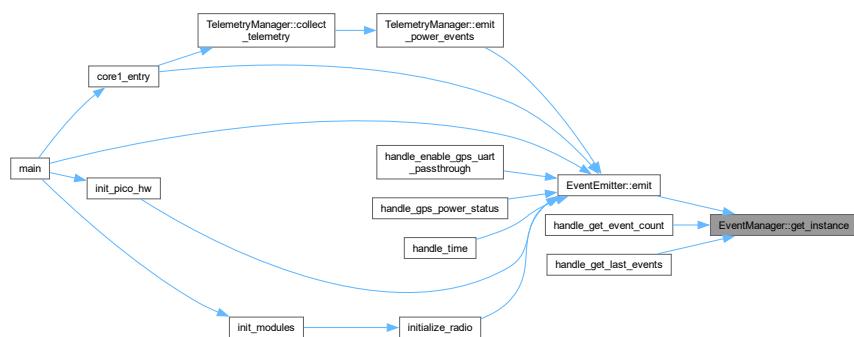
A reference to the singleton instance.

Definition at line 220 of file [event_manager.h](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.11.3.3 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 249 of file [event_manager.h](#).

8.11.4 Member Data Documentation

8.11.4.1 events

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

Definition at line 196 of file [event_manager.h](#).

8.11.4.2 eventCount

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

Definition at line 197 of file [event_manager.h](#).

8.11.4.3 writeIndex

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

Definition at line 198 of file [event_manager.h](#).

8.11.4.4 eventMutex

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

Definition at line 199 of file [event_manager.h](#).

8.11.4.5 nextEventId

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

Definition at line 200 of file [event_manager.h](#).

8.11.4.6 eventsSinceFlush

```
size_t EventManager::eventsSinceFlush [private]
```

Definition at line 201 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](#)

8.12 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`

8.12.1 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 143 of file [protocol.h](#).

8.12.2 Member Data Documentation

8.12.2.1 `header`

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

Definition at line 144 of file [protocol.h](#).

8.12.2.2 `direction`

```
uint8_t Frame::direction
```

Definition at line 145 of file [protocol.h](#).

8.12.2.3 `operationType`

```
OperationType Frame::operationType
```

Definition at line 146 of file [protocol.h](#).

8.12.2.4 group

```
uint8_t Frame::group
```

Definition at line 147 of file [protocol.h](#).

8.12.2.5 command

```
uint8_t Frame::command
```

Definition at line 148 of file [protocol.h](#).

8.12.2.6 value

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

Definition at line 149 of file [protocol.h](#).

8.12.2.7 unit

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

Definition at line 150 of file [protocol.h](#).

8.12.2.8 footer

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

Definition at line 151 of file [protocol.h](#).

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

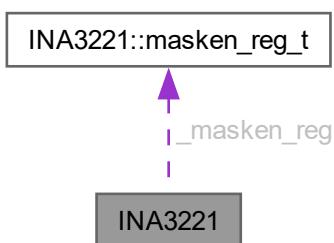
- lib/comms/[protocol.h](#)

8.13 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 set_mode_continuous ()`
Set device to continuous measurement mode.
- `void set_mode_triggered ()`
Set device to triggered measurement mode.
- `void set_averaging_mode (ina3221_avg_mode_t mode)`
Set the averaging mode for measurements.
- `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_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`

8.13.1 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 86 of file [INA3221.h](#).

8.13.2 Member Function Documentation

8.13.2.1 `_read()`

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

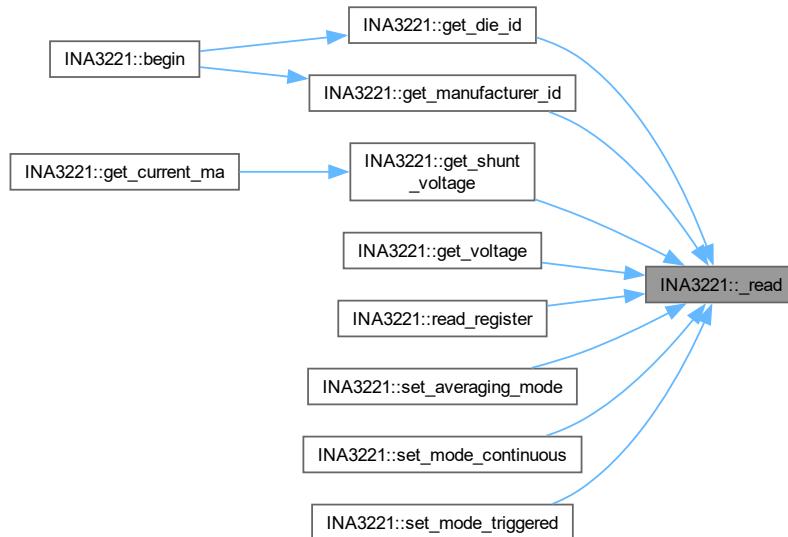
Read a 16-bit register from the device.

Parameters

<i>reg</i>	Register address
<i>val</i>	Pointer to store the read value

Definition at line 242 of file [INA3221.cpp](#).

Here is the caller graph for this function:



8.13.2.2 `_write()`

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

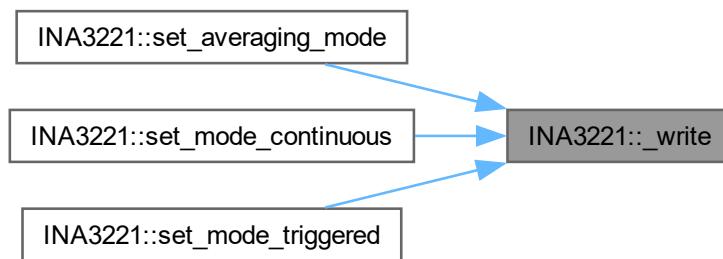
Write a 16-bit value to a register.

Parameters

<i>reg</i>	Register address
<i>val</i>	Pointer to the value to write

Definition at line 268 of file [INA3221.cpp](#).

Here is the caller graph for this function:



8.13.3 Member Data Documentation

8.13.3.1 _i2c_addr

`ina3221_addr_t INA3221::_i2c_addr [private]`

Definition at line 127 of file [INA3221.h](#).

8.13.3.2 _i2c

`i2c_inst_t* INA3221::_i2c [private]`

Definition at line 128 of file [INA3221.h](#).

8.13.3.3 _shuntRes

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

Definition at line 131 of file [INA3221.h](#).

8.13.3.4 _filterRes

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

Definition at line 134 of file [INA3221.h](#).

8.13.3.5 _masken_reg

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

Definition at line 137 of file [INA3221.h](#).

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

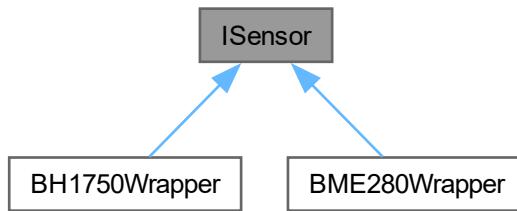
- lib/powerman/INA3221/[INA3221.h](#)
- lib/powerman/INA3221/[INA3221.cpp](#)

8.14 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 \(SensorDataTypelIdentifier type\)=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.

8.14.1 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](#).

8.14.2 Constructor & Destructor Documentation

8.14.2.1 ~ISensor()

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

Virtual destructor.

Ensures proper cleanup of derived classes.

8.14.3 Member Function Documentation

8.14.3.1 init()

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

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implemented in [BH1750Wrapper](#), and [BME280Wrapper](#).

8.14.3.2 read_data()

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

Reads data from the sensor.

Parameters

in	<i>type</i>	Data type to read.
----	-------------	--------------------

Returns

The sensor data.

Implemented in [BH1750Wrapper](#), and [BME280Wrapper](#).

8.14.3.3 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](#).

8.14.3.4 get_type()

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

Gets the sensor type.

Returns

The sensor type.

Implemented in [BH1750Wrapper](#), and [BME280Wrapper](#).

8.14.3.5 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](#).

8.14.3.6 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](#)

8.15 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`

8.15.1 Detailed Description

Mask/Enable register bit fields.

Definition at line 107 of file [INA3221.h](#).

8.15.2 Member Data Documentation

8.15.2.1 conv_ready

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

Definition at line 108 of file [INA3221.h](#).

8.15.2.2 timing_ctrl_alert

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

Definition at line 109 of file [INA3221.h](#).

8.15.2.3 pwr_valid_alert

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

Definition at line 110 of file [INA3221.h](#).

8.15.2.4 warn_alert_ch3

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

Definition at line 111 of file [INA3221.h](#).

8.15.2.5 warn_alert_ch2

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

Definition at line 112 of file [INA3221.h](#).

8.15.2.6 warn_alert_ch1

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

Definition at line 113 of file [INA3221.h](#).

8.15.2.7 shunt_sum_alert

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

Definition at line 114 of file [INA3221.h](#).

8.15.2.8 crit_alert_ch3

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

Definition at line 115 of file [INA3221.h](#).

8.15.2.9 crit_alert_ch2

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

Definition at line 116 of file [INA3221.h](#).

8.15.2.10 crit_alert_ch1

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

Definition at line 117 of file [INA3221.h](#).

8.15.2.11 crit_alert_latch_en

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

Definition at line 118 of file [INA3221.h](#).

8.15.2.12 warn_alert_latch_en

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

Definition at line 119 of file [INA3221.h](#).

8.15.2.13 shunt_sum_en_ch3

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

Definition at line 120 of file [INA3221.h](#).

8.15.2.14 shunt_sum_en_ch2

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

Definition at line 121 of file [INA3221.h](#).

8.15.2.15 shunt_sum_en_ch1

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

Definition at line 122 of file [INA3221.h](#).

8.15.2.16 reserved

```
uint16_t INA3221::masken_reg_t::reserved
```

Definition at line 123 of file [INA3221.h](#).

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

- lib/powerman/INA3221/[INA3221.h](#)

8.16 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.

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.

8.16.1 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 32 of file `NMEA_data.h`.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 NMEAData() [1/2]

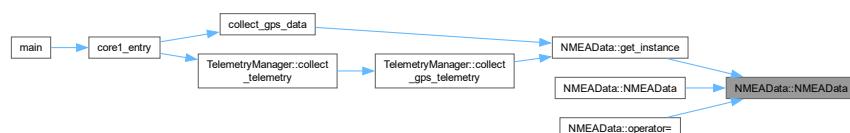
```
NMEAData::NMEAData () [inline], [private]
```

Private constructor for the singleton pattern.

Initializes the mutexes.

Definition at line 47 of file `NMEA_data.h`.

Here is the caller graph for this function:



8.16.2.2 NMEAData() [2/2]

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

Deleted copy constructor to prevent copying.

Here is the call graph for this function:



8.16.3 Member Function Documentation

8.16.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

Here is the call graph for this function:



8.16.3.2 get_instance()

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

Gets the singleton instance of the [NMEAData](#) class.

Returns

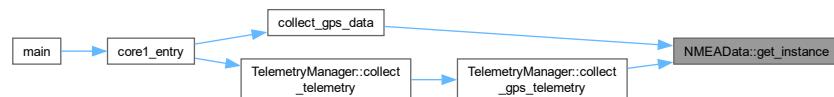
A reference to the singleton instance.

Definition at line 66 of file [NMEA_data.h](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**8.16.3.3 update_rmc_tokens()**

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

Updates the RMC tokens with new data.

Parameters

in	<i>tokens</i>	Vector of strings representing the RMC tokens.
----	---------------	--

Definition at line 75 of file [NMEA_data.h](#).

Here is the caller graph for this function:

**8.16.3.4 update_gga_tokens()**

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

Updates the GGA tokens with new data.

Parameters

in	<i>tokens</i>	Vector of strings representing the GGA tokens.
----	---------------	--

Definition at line 85 of file [NMEA_data.h](#).

Here is the caller graph for this function:



8.16.3.5 `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 95 of file [NMEA_data.h](#).

8.16.3.6 `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 106 of file [NMEA_data.h](#).

8.16.4 Member Data Documentation

8.16.4.1 `rmc_tokens_`

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

Vector of tokens from the most recent RMC sentence.

Definition at line 35 of file [NMEA_data.h](#).

8.16.4.2 gga_tokens_

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

Vector of tokens from the most recent GGA sentence.

Definition at line 37 of file [NMEA_data.h](#).

8.16.4.3 rmc_mutex_

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

Mutex for thread-safe access to the RMC tokens.

Definition at line 39 of file [NMEA_data.h](#).

8.16.4.4 gga_mutex_

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

Mutex for thread-safe access to the GGA tokens.

Definition at line 41 of file [NMEA_data.h](#).

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

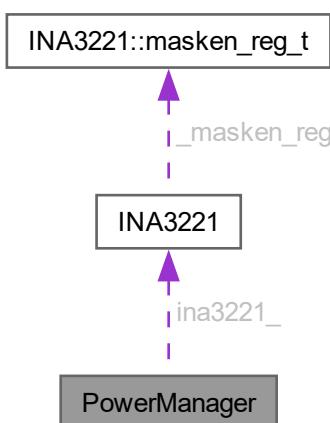
- lib/location/NMEA/[NMEA_data.h](#)

8.17 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.`
- `float get_voltage_solar ()`
`Gets solar voltage.`
- `void configure (ina3221_op_mode_t op_mode, ina3221_avg_mode_t avg_mode)`
`Configures the INA3221.`

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`.

8.17.1 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`.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 PowerManager() [1/2]

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

Constructor for the `PowerManager` class.

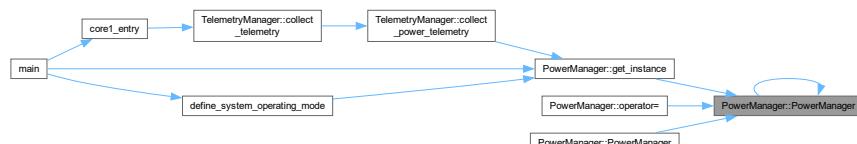
Parameters

in	<code>i2c</code>	I2C instance to use for communication with the <code>INA3221</code> .
----	------------------	---

Here is the call graph for this function:



Here is the caller graph for this function:



8.17.2.2 PowerManager() [2/2]

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

Deleted copy constructor to prevent copying.

Here is the call graph for this function:



8.17.3 Member Function Documentation

8.17.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

Here is the call graph for this function:



8.17.4 Member Data Documentation

8.17.4.1 SOLAR_CURRENT_THRESHOLD

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

Solar current threshold in milliamperes.

Definition at line 108 of file [PowerManager.h](#).

8.17.4.2 USB_CURRENT_THRESHOLD

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

USB current threshold in milliamperes.

Definition at line 110 of file [PowerManager.h](#).

8.17.4.3 BATTERY_LOW_THRESHOLD

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

Low voltage threshold in volts.

Definition at line 112 of file [PowerManager.h](#).

8.17.4.4 BATTERY_FULL_THRESHOLD

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

Overcharge voltage threshold in volts.

Definition at line 114 of file [PowerManager.h](#).

8.17.4.5 ina3221_

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

[INA3221](#) instance for power monitoring.

Definition at line 118 of file [PowerManager.h](#).

8.17.4.6 initialized_

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

Flag indicating if the [PowerManager](#) is initialized.

Definition at line 120 of file [PowerManager.h](#).

8.17.4.7 powerman_mutex_

```
recursive_mutex_t PowerManager::powerman_mutex_ [private]
```

Mutex for thread-safe access to the [PowerManager](#).

Definition at line 122 of file [PowerManager.h](#).

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

- lib/powerman/[PowerManager.h](#)
- lib/powerman/[PowerManager.cpp](#)

8.18 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](#)

8.18.1 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 [111](#) of file [telemetry_manager.h](#).

8.18.2 Member Data Documentation

8.18.2.1 timestamp

```
uint32_t SensorDataRecord::timestamp
```

Unix timestamp of the record

Definition at line [112](#) of file [telemetry_manager.h](#).

8.18.2.2 temperature

```
float SensorDataRecord::temperature
```

Temperature in degrees Celsius

Definition at line [113](#) of file [telemetry_manager.h](#).

8.18.2.3 pressure

```
float SensorDataRecord::pressure
```

Pressure in hPa

Definition at line 114 of file [telemetry_manager.h](#).

8.18.2.4 humidity

```
float SensorDataRecord::humidity
```

Relative humidity in %

Definition at line 115 of file [telemetry_manager.h](#).

8.18.2.5 light

```
float SensorDataRecord::light
```

Light intensity in lux

Definition at line 116 of file [telemetry_manager.h](#).

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

- lib/telemetry/[telemetry_manager.h](#)

8.19 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](#), [SensorDataTypelIdentifier](#) [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.

8.19.1 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`.

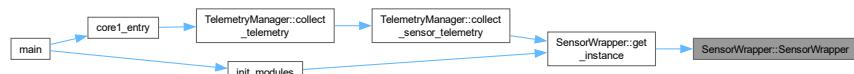
8.19.2 Constructor & Destructor Documentation

8.19.2.1 SensorWrapper()

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

Private constructor for the singleton pattern.

Here is the caller graph for this function:



8.19.3 Member Function Documentation

8.19.3.1 get_instance()

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

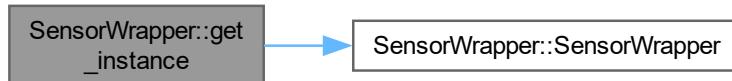
Gets the singleton instance of the [SensorWrapper](#) class.

Returns

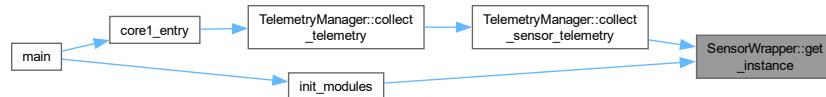
A reference to the singleton instance.

Definition at line 122 of file [ISensor.h](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.19.3.2 scan_connected_sensors()

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

Scans for connected sensors.

Parameters

in	<i>i2c</i>	I2C instance to use for scanning.
----	------------	-----------------------------------

Returns

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

8.19.3.3 `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.

8.19.4 Member Data Documentation

8.19.4.1 `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](#)

8.20 SystemStateManager Class Reference

Manages the system state of the Kabisat firmware.

```
#include <system_state_manager.h>
```

Public Member Functions

- bool [is_bootloader_reset_pending](#) () const
Checks if a bootloader reset is pending.
- void [set_bootloader_reset_pending](#) (bool pending)
Sets whether a bootloader reset is pending.
- bool [is_gps_collection_paused](#) () const
Checks if GPS collection is paused.
- void [set_gps_collection_paused](#) (bool paused)
Sets whether GPS collection is paused.
- bool [is_sd_card_mounted](#) () const
Checks if the SD card is mounted.
- void [set_sd_card_mounted](#) (bool mounted)
Sets whether the SD card is mounted.
- [VerbosityLevel get_uart_verbosity](#) () const
Gets the UART verbosity level.
- void [set_uart_verbosity](#) ([VerbosityLevel](#) level)
Sets the UART verbosity level.

- `bool is_radio_init_ok () const`
Checks if the radio initialization was successful.
- `void set_radio_init_ok (bool status)`
Sets whether the radio initialization was successful.
- `bool is_light_sensor_init_ok () const`
Checks if the light sensor initialization was successful.
- `void set_light_sensor_init_ok (bool status)`
Sets whether the light sensor initialization was successful.
- `bool is_env_sensor_init_ok () const`
Checks if the environment sensor initialization was successful.
- `void set_env_sensor_init_ok (bool status)`
Sets whether the environment sensor initialization was successful.
- `SystemOperatingMode get_operating_mode () const`
Gets the system operating mode.
- `void set_operating_mode (SystemOperatingMode mode)`
Sets the system operating mode.

Static Public Member Functions

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

Private Member Functions

- `SystemStateManager ()`
Private constructor for the singleton pattern.
- `SystemStateManager (const SystemStateManager &) = delete`
Deleted copy constructor to prevent copying.
- `SystemStateManager & operator= (const SystemStateManager &) = delete`
Deleted assignment operator to prevent assignment.

Private Attributes

- `bool pending_bootloader_reset`
Flag indicating whether a bootloader reset is pending.
- `bool gps_collection_paused`
Flag indicating whether GPS collection is paused.
- `bool sd_card_mounted`
Flag indicating whether the SD card is mounted.
- `VerbosityLevel uart_verbosity`
The UART verbosity level.
- `bool sd_card_init_status`
Flag indicating whether the SD card initialization was successful.
- `bool radio_init_status`
Flag indicating whether the radio initialization was successful.
- `bool light_sensor_init_status`
Flag indicating whether the light sensor initialization was successful.
- `bool env_sensor_init_status`
Flag indicating whether the environment sensor initialization was successful.
- `SystemOperatingMode system_operating_mode`
The system operating mode.
- `recursive_mutex_t mutex`
Mutex for thread-safe access to the system state.

8.20.1 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 40 of file [system_state_manager.h](#).

8.20.2 Constructor & Destructor Documentation

8.20.2.1 SystemStateManager() [1/2]

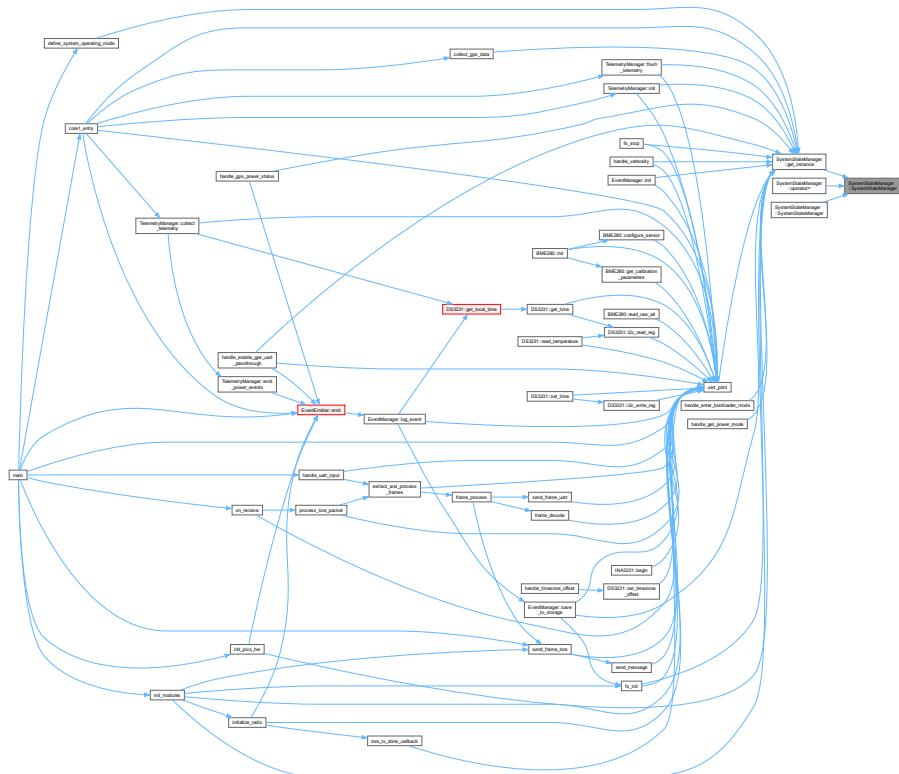
```
SystemStateManager::SystemStateManager () [inline], [private]
```

Private constructor for the singleton pattern.

Initializes the system state and mutex.

Definition at line 67 of file [system_state_manager.h](#).

Here is the caller graph for this function:

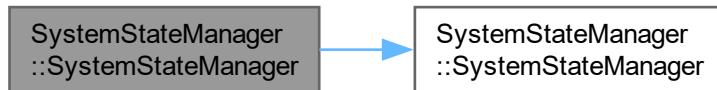


8.20.2.2 SystemStateManager() [2/2]

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

Deleted copy constructor to prevent copying.

Here is the call graph for this function:



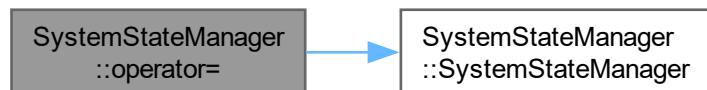
8.20.3 Member Function Documentation

8.20.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

Here is the call graph for this function:



8.20.3.2 get_instance()

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

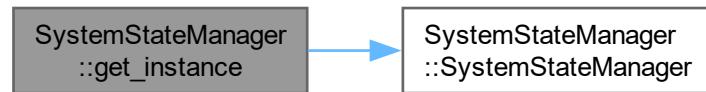
Gets the singleton instance of the [SystemStateManager](#) class.

Returns

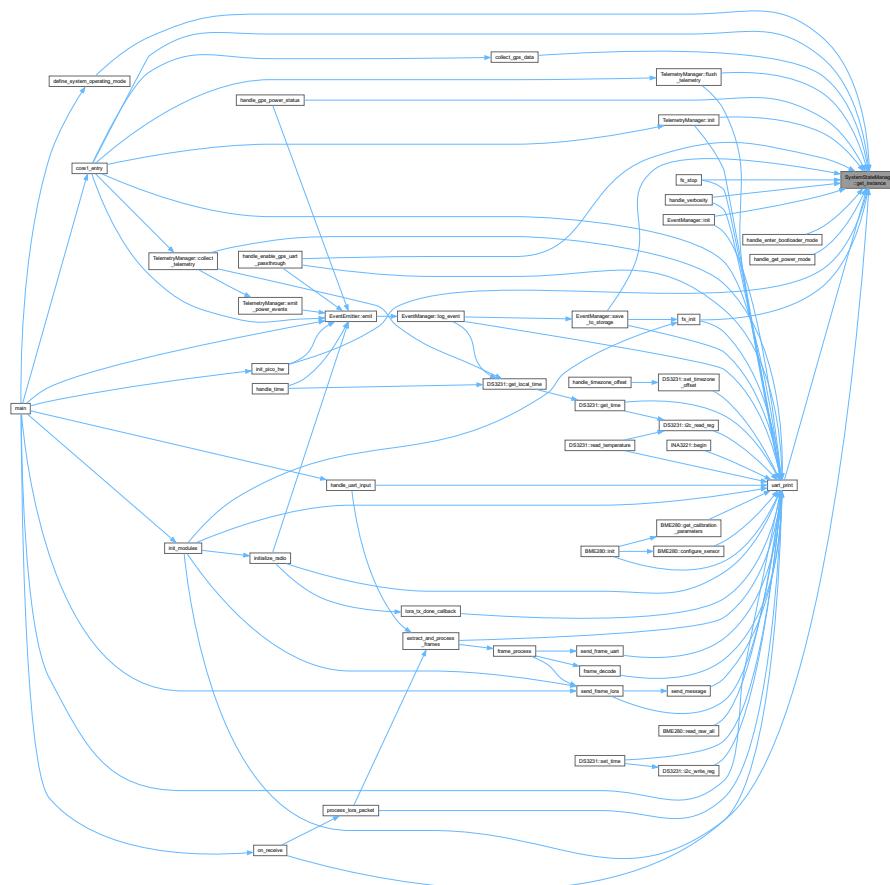
A reference to the singleton instance.

Definition at line 95 of file [system_state_manager.h](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.20.3.3 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 104 of file [system_state_manager.h](#).

8.20.3.4 set_bootloader_reset_pending()

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

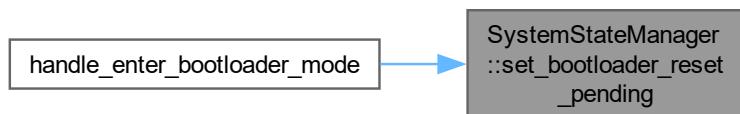
Sets whether a bootloader reset is pending.

Parameters

in	<i>pending</i>	True if a bootloader reset is pending, false otherwise.
----	----------------	---

Definition at line 115 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.3.5 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 125 of file [system_state_manager.h](#).

8.20.3.6 set_gps_collection_paused()

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

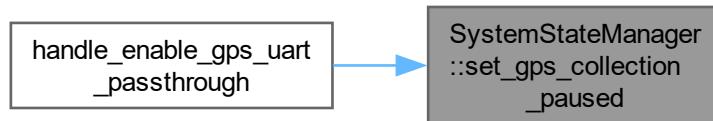
Sets whether GPS collection is paused.

Parameters

in	<i>paused</i>	True if GPS collection is paused, false otherwise.
----	---------------	--

Definition at line 136 of file [system_state_manager.h](#).

Here is the caller graph for this function:

**8.20.3.7 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 146 of file [system_state_manager.h](#).

8.20.3.8 set_sd_card_mounted()

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

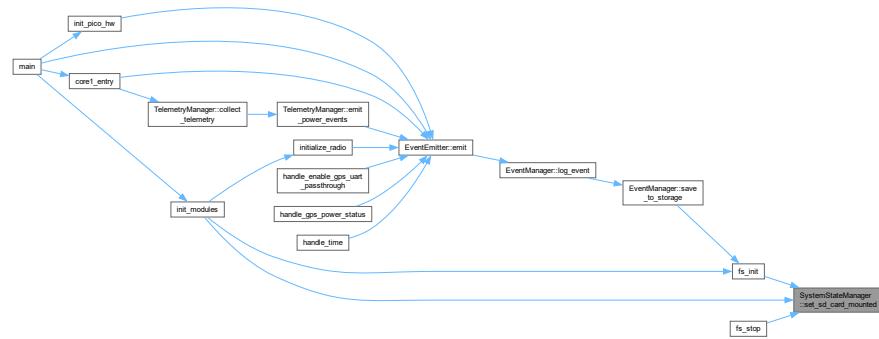
Sets whether the SD card is mounted.

Parameters

in	<i>mounted</i>	True if the SD card is mounted, false otherwise.
----	----------------	--

Definition at line 157 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.3.9 get_uart_verbosity()

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

Gets the UART verbosity level.

Returns

The UART verbosity level.

Definition at line 167 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.3.10 set_uart_verbosity()

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

Sets the UART verbosity level.

Parameters

in	<i>level</i>	The UART verbosity level.
----	--------------	---------------------------

Definition at line 178 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.3.11 is_radio_init_ok()

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

Checks if the radio initialization was successful.

Returns

True if the radio initialization was successful, false otherwise.

Definition at line 188 of file [system_state_manager.h](#).

8.20.3.12 set_radio_init_ok()

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

Sets whether the radio initialization was successful.

Parameters

in	<i>status</i>	True if the radio initialization was successful, false otherwise.
----	---------------	---

Definition at line 199 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.3.13 is_light_sensor_init_ok()

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

Checks if the light sensor initialization was successful.

Returns

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

Definition at line 209 of file [system_state_manager.h](#).

8.20.3.14 set_light_sensor_init_ok()

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

Sets whether the light sensor initialization was successful.

Parameters

in	<i>status</i>	True if the light sensor initialization was successful, false otherwise.
----	---------------	--

Definition at line 220 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.3.15 is_env_sensor_init_ok()

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

Checks if the environment sensor initialization was successful.

Returns

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

Definition at line 230 of file [system_state_manager.h](#).

8.20.3.16 set_env_sensor_init_ok()

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

Sets whether the environment sensor initialization was successful.

Parameters

<code>in</code>	<code>status</code>	True if the environment sensor initialization was successful, false otherwise.
-----------------	---------------------	--

Definition at line 241 of file [system_state_manager.h](#).

Here is the caller graph for this function:

**8.20.3.17 get_operating_mode()**

`SystemOperatingMode SystemStateManager::get_operating_mode () const [inline]`

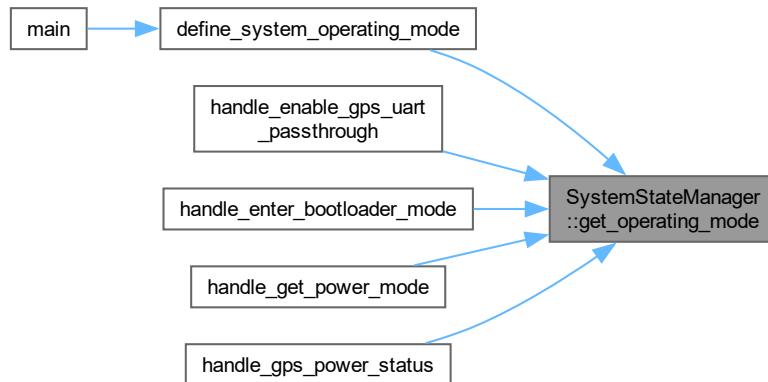
Gets the system operating mode.

Returns

The system operating mode.

Definition at line 251 of file [system_state_manager.h](#).

Here is the caller graph for this function:

**8.20.3.18 set_operating_mode()**

```
void SystemStateManager::set_operating_mode (
    SystemOperatingMode mode) [inline]
```

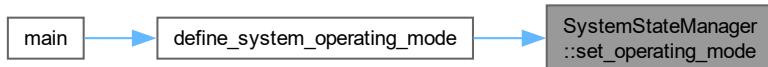
Sets the system operating mode.

Parameters

in	<i>mode</i>	The system operating mode.
----	-------------	----------------------------

Definition at line 262 of file [system_state_manager.h](#).

Here is the caller graph for this function:



8.20.4 Member Data Documentation

8.20.4.1 pending_bootloader_reset

```
bool SystemStateManager::pending_bootloader_reset [private]
```

Flag indicating whether a bootloader reset is pending.

Definition at line 43 of file [system_state_manager.h](#).

8.20.4.2 gps_collection_paused

```
bool SystemStateManager::gps_collection_paused [private]
```

Flag indicating whether GPS collection is paused.

Definition at line 45 of file [system_state_manager.h](#).

8.20.4.3 sd_card_mounted

```
bool SystemStateManager::sd_card_mounted [private]
```

Flag indicating whether the SD card is mounted.

Definition at line 47 of file [system_state_manager.h](#).

8.20.4.4 uart_verbosity

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

The UART verbosity level.

Definition at line 49 of file [system_state_manager.h](#).

8.20.4.5 `sd_card_init_status`

```
bool SystemStateManager::sd_card_init_status [private]
```

Flag indicating whether the SD card initialization was successful.

Definition at line 51 of file [system_state_manager.h](#).

8.20.4.6 `radio_init_status`

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

Flag indicating whether the radio initialization was successful.

Definition at line 53 of file [system_state_manager.h](#).

8.20.4.7 `light_sensor_init_status`

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

Flag indicating whether the light sensor initialization was successful.

Definition at line 55 of file [system_state_manager.h](#).

8.20.4.8 `env_sensor_init_status`

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

Flag indicating whether the environment sensor initialization was successful.

Definition at line 57 of file [system_state_manager.h](#).

8.20.4.9 `system_operating_mode`

```
SystemOperatingMode SystemStateManager::system_operating_mode [private]
```

The system operating mode.

Definition at line 59 of file [system_state_manager.h](#).

8.20.4.10 `mutex_`

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

Mutex for thread-safe access to the system state.

Definition at line 61 of file [system_state_manager.h](#).

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

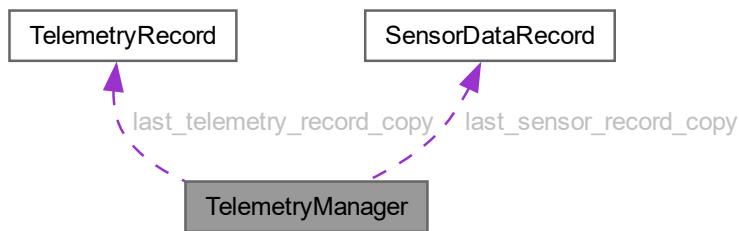
- lib/[system_state_manager.h](#)

8.21 TelemetryManager Class Reference

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

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

Collaboration diagram for TelemetryManager:



Public Member Functions

- bool `init ()`
Initialize the telemetry system.
- bool `collect_telemetry ()`
Collect telemetry data from sensors and power subsystems.
- void `collect_power_telemetry (TelemetryRecord &record)`
Collects power subsystem telemetry data.
- void `emit_power_events (float battery_voltage, float charge_current_usb, float charge_current_solar, float discharge_current)`
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.
- size_t `get_telemetry_buffer_count () const`
- size_t `get_telemetry_buffer_write_index () const`

Static Public Member Functions

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

Static Public Attributes

- static constexpr int `TELEMETRY_BUFFER_SIZE = 20`

Private Member Functions

- `TelemetryManager ()`
- `~TelemetryManager ()=default`

Private Attributes

- `uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS`
Current sampling interval in milliseconds.
- `uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD`
Current flush threshold (number of records that triggers a flush)
- `std::array< TelemetryRecord, TELEMETRY_BUFFER_SIZE > telemetry_buffer`
Circular buffer for telemetry records.
- `size_t telemetry_buffer_count = 0`
- `size_t telemetry_buffer_write_index = 0`
- `std::array< SensorDataRecord, TELEMETRY_BUFFER_SIZE > sensor_data_buffer`
Circular buffer for sensor data records.
- `TelemetryRecord last_telemetry_record_copy`
Last record copies for retrieval.
- `SensorDataRecord last_sensor_record_copy`
- `mutex_t telemetry_mutex`
Mutex for thread-safe access to the telemetry buffer.

Static Private Attributes

- static constexpr uint32_t `DEFAULT_SAMPLE_INTERVAL_MS = 1000`
Default sampling interval in milliseconds.
- static constexpr uint32_t `DEFAULT_FLUSH_THRESHOLD = 10`
Default number of records to collect before flushing to storage.

8.21.1 Detailed Description

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

This class implements a singleton pattern to provide a single point of access for managing telemetry data. It handles the collection of data from various subsystems, stores the data in circular buffers, and provides methods for flushing the data to persistent storage and retrieving the last recorded data.

Definition at line 146 of file `telemetry_manager.h`.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 ~TelemetryManager()

```
TelemetryManager::~TelemetryManager () [private], [default]
```

8.21.3 Member Function Documentation

8.21.3.1 get_instance()

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

Gets the singleton instance of the [TelemetryManager](#) class.

Returns

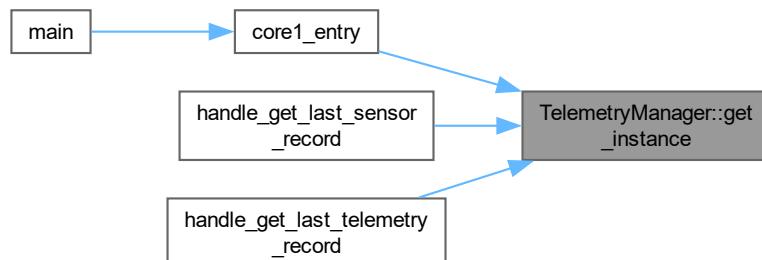
A reference to the singleton instance.

Definition at line 152 of file [telemetry_manager.h](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.21.3.2 flush_sensor_data()

```
bool TelemetryManager::flush_sensor_data ()
```

Save buffered sensor data to storage.

Returns

True if data was successfully saved

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

8.21.3.3 get_telemetry_buffer_count()

```
size_t TelemetryManager::get_telemetry_buffer_count () const [inline]
```

Definition at line 252 of file [telemetry_manager.h](#).

8.21.3.4 get_telemetry_buffer_write_index()

```
size_t TelemetryManager::get_telemetry_buffer_write_index () const [inline]
```

Definition at line 253 of file [telemetry_manager.h](#).

8.21.4 Member Data Documentation

8.21.4.1 TELEMTRY_BUFFER_SIZE

```
int TelemetryManager::TELEMTRY_BUFFER_SIZE = 20 [static], [constexpr]
```

Definition at line 250 of file [telemetry_manager.h](#).

8.21.4.2 DEFAULT_SAMPLE_INTERVAL_MS

```
uint32_t TelemetryManager::DEFAULT_SAMPLE_INTERVAL_MS = 1000 [static], [constexpr], [private]
```

Current sampling interval in milliseconds.

Definition at line 262 of file [telemetry_manager.h](#).

8.21.4.3 DEFAULT_FLUSH_THRESHOLD

```
uint32_t TelemetryManager::DEFAULT_FLUSH_THRESHOLD = 10 [static], [constexpr], [private]
```

Default number of records to collect before flushing to storage.

Definition at line 267 of file [telemetry_manager.h](#).

8.21.4.4 sample_interval_ms

```
uint32_t TelemetryManager::sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS [private]
```

Definition at line 269 of file [telemetry_manager.h](#).

8.21.4.5 flush_threshold

```
uint32_t TelemetryManager::flush_threshold = DEFAULT_FLUSH_THRESHOLD [private]
```

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

Definition at line 274 of file [telemetry_manager.h](#).

8.21.4.6 telemetry_buffer

```
std::array<TelemetryRecord, TELEMETRY_BUFFER_SIZE> TelemetryManager::telemetry_buffer [private]
```

Circular buffer for telemetry records.

Definition at line 278 of file [telemetry_manager.h](#).

8.21.4.7 telemetry_buffer_count

```
size_t TelemetryManager::telemetry_buffer_count = 0 [private]
```

Definition at line 279 of file [telemetry_manager.h](#).

8.21.4.8 telemetry_buffer_write_index

```
size_t TelemetryManager::telemetry_buffer_write_index = 0 [private]
```

Definition at line 280 of file [telemetry_manager.h](#).

8.21.4.9 sensor_data_buffer

```
std::array<SensorDataRecord, TELEMETRY_BUFFER_SIZE> TelemetryManager::sensor_data_buffer [private]
```

Circular buffer for sensor data records.

Definition at line 285 of file [telemetry_manager.h](#).

8.21.4.10 last_telemetry_record_copy

```
TelemetryRecord TelemetryManager::last_telemetry_record_copy [private]
```

Last record copies for retrieval.

Definition at line 290 of file [telemetry_manager.h](#).

8.21.4.11 `last_sensor_record_copy`

```
SensorDataRecord TelemetryManager::last_sensor_record_copy [private]
```

Definition at line 291 of file [telemetry_manager.h](#).

8.21.4.12 `telemetry_mutex`

```
mutex_t TelemetryManager::telemetry_mutex [private]
```

Mutex for thread-safe access to the telemetry buffer.

Definition at line 296 of file [telemetry_manager.h](#).

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

- lib/telemetry/[telemetry_manager.h](#)
- lib/telemetry/[telemetry_manager.cpp](#)

8.22 TelemetryRecord Struct Reference

Structure representing a single telemetry data point.

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

Public Member Functions

- std::string [to_csv \(\) const](#)
Converts the telemetry record to a CSV string.

Public Attributes

- uint32_t [timestamp](#)
- std::string [build_version](#)
- float [battery_voltage](#)
- float [system_voltage](#)
- float [charge_current_usb](#)
- float [charge_current_solar](#)
- float [discharge_current](#)
- std::string [time](#)
- std::string [latitude](#)
- std::string [lat_dir](#)
- std::string [longitude](#)
- std::string [lon_dir](#)
- std::string [speed](#)
- std::string [course](#)
- std::string [date](#)
- std::string [fix_quality](#)
- std::string [satellites](#)
- std::string [altitude](#)

8.22.1 Detailed Description

Structure representing a single telemetry data point.

Contains all measurements from power subsystem, sensors, and GPS data collected at a specific point in time

Definition at line 44 of file [telemetry_manager.h](#).

8.22.2 Member Data Documentation

8.22.2.1 timestamp

```
uint32_t TelemetryRecord::timestamp
```

Unix timestamp of the record

Definition at line 45 of file [telemetry_manager.h](#).

8.22.2.2 build_version

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

Build version of the firmware

Definition at line 47 of file [telemetry_manager.h](#).

8.22.2.3 battery_voltage

```
float TelemetryRecord::battery_voltage
```

Battery voltage in volts

Definition at line 50 of file [telemetry_manager.h](#).

8.22.2.4 system_voltage

```
float TelemetryRecord::system_voltage
```

System 5V rail voltage in volts

Definition at line 51 of file [telemetry_manager.h](#).

8.22.2.5 charge_current_usb

```
float TelemetryRecord::charge_current_usb
```

USB charging current in mA

Definition at line 52 of file [telemetry_manager.h](#).

8.22.2.6 charge_current_solar

```
float TelemetryRecord::charge_current_solar
```

Solar charging current in mA

Definition at line 53 of file [telemetry_manager.h](#).

8.22.2.7 discharge_current

```
float TelemetryRecord::discharge_current
```

Battery discharge current in mA

Definition at line 54 of file [telemetry_manager.h](#).

8.22.2.8 time

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

UTC time from GPS

Definition at line 57 of file [telemetry_manager.h](#).

8.22.2.9 latitude

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

Latitude from GPS

Definition at line 58 of file [telemetry_manager.h](#).

8.22.2.10 lat_dir

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

N/S latitude direction

Definition at line 59 of file [telemetry_manager.h](#).

8.22.2.11 longitude

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

Longitude from GPS

Definition at line 60 of file [telemetry_manager.h](#).

8.22.2.12 lon_dir

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

E/W longitude direction

Definition at line 61 of file [telemetry_manager.h](#).

8.22.2.13 speed

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

Speed in knots

Definition at line 62 of file [telemetry_manager.h](#).

8.22.2.14 course

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

Course in degrees

Definition at line 63 of file [telemetry_manager.h](#).

8.22.2.15 date

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

Date from GPS

Definition at line 64 of file [telemetry_manager.h](#).

8.22.2.16 fix_quality

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

GPS fix quality

Definition at line 67 of file [telemetry_manager.h](#).

8.22.2.17 satellites

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

Number of satellites in view

Definition at line 68 of file [telemetry_manager.h](#).

8.22.2.18 altitude

```
std::string TelemetryRecord::altitude
```

Altitude in meters

Definition at line 69 of file [telemetry_manager.h](#).

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

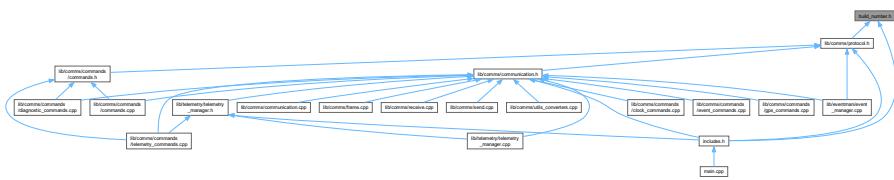
- lib/telemetry/[telemetry_manager.h](#)

Chapter 9

File Documentation

9.1 build_number.h File Reference

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



Macros

- `#define BUILD_NUMBER 541`

9.1.1 Macro Definition Documentation

9.1.1.1 BUILD_NUMBER

```
#define BUILD_NUMBER 541
```

Definition at line 6 of file [build_number.h](#).

9.2 build_number.h

[Go to the documentation of this file.](#)

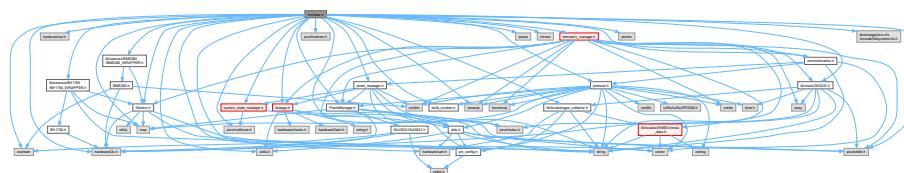
```
00001 //This file is automatically generated by build_number.cmake
00002
00003 #ifndef CMAKE_BUILD_NUMBER_HEADER
00004 #define CMAKE_BUILD_NUMBER_HEADER
00005
00006 #define BUILD_NUMBER 541
00007
00008 #endif
```

9.3 credits.md File Reference

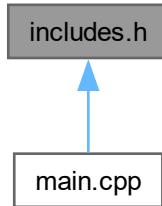
9.4 includes.h File Reference

```
#include <stdio.h>
#include "pico/stl.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:



9.5 includes.h

[Go to the documentation of this file.](#)

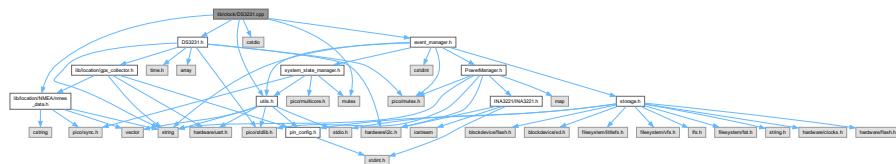
```
00001 #ifndef INCLUDES_H
00002 #define INCLUDES_H
00003
00004 #include <stdio.h>
00005 #include "pico/stlplib.h"
00006 #include "hardware/spi.h"
00007 #include "hardware/i2c.h"
00008 #include "hardware/uart.h"
00009 #include "pico/multicore.h"
00010 #include "event_manager.h"
00011 #include "lib/powerman/PowerManager.h"
00012 #include <pico/bootrom.h>
00013
00014 #include "ISensor.h"
00015 #include "lib/sensors/BH1750/BH1750_WRAPPER.h"
00016 #include "lib/sensors/BME280/BME280_WRAPPER.h"
00017 #include "lib/clock/DS3231.h"
00018 #include <iostream>
00019 #include <iomanip>
00020 #include <queue>
00021 #include <chrono>
00022 #include "protocol.h"
00023 #include <atomic>
00024 #include <iostream>
00025 #include <map>
00026 #include "pin_config.h"
00027 #include "utils.h"
00028 #include "communication.h"
00029 #include "build_number.h"
00030 #include "lib/location/gps_collector.h"
00031 #include "lib/storage/storage.h"
00032 #include "lib/storage/pico-vfs/include/filesystem/vfs.h"
00033 #include "telemetry_manager.h"
00034 #include "system_state_manager.h"
00035
00036 #endif
```

9.6 lib/clock/DS3231.cpp File Reference

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

```
#include "NMEA_data.
```

Include dependency graph for DS3231.cpp:



9.7 DS3231.cpp

[Go to the documentation of this file.](#)

```

00001 #include "DS3231.h"
00002 #include "utils.h"
00003 #include <cstdio>
00004 #include <mutex>
00005 #include "event_manager.h"
00006 #include "NMEA_data.h"
00007
00013
00023 DS3231::DS3231() : i2c(MAIN_I2C_PORT), ds3231_addr(DS3231_DEVICE_ADDRESS) {
00024     recursive_mutex_init(&clock_mutex);
00025 }
00026
00027
00038 DS3231& DS3231::get_instance() {
00039     static DS3231 instance;
00040     return instance;
00041 }
00042
00047 time_t DS3231::get_time() {
00048     uint8_t time_data[7];
00049     if (I2c_read_reg(DS3231_SECONDS_REG, 7, time_data) != 0) {
00050         uart_print("Failed to read time from RTC", VerboseLevel::ERROR);
00051         return -1;
00052     }
00053
00054     struct tm timeinfo = {};
00055     // Convert BCD to binary and fill tm structure
00056     timeinfo.tm_sec = ((time_data[0] >> 4) * 10) + (time_data[0] & 0x0F);
00057     timeinfo.tm_min = ((time_data[1] >> 4) * 10) + (time_data[1] & 0x0F);
00058     timeinfo.tm_hour = ((time_data[2] >> 4) * 10) + (time_data[2] & 0x0F);
00059     timeinfo.tm_mday = ((time_data[4] >> 4) * 10) + (time_data[4] & 0x0F);
00060     timeinfo.tm_mon = (((time_data[5] & 0x1F) >> 4) * 10) + (time_data[5] & 0x0F) - 1;
00061     timeinfo.tm_year = (((time_data[6] >> 4) * 10) + (time_data[6] & 0x0F) + 100;
00062     timeinfo.tm_isdst = 0;
00063
00064     time_t unix_time = mktime(&timeinfo);
00065     if (unix_time == -1) {
00066         uart_print("Failed to convert RTC time", VerboseLevel::ERROR);
00067         return -1;
00068     }
00069
00070     return unix_time;
00071 }
00072
00073
00079 int DS3231::set_time(time_t unix_time) {
00080     struct tm* timeinfo = gmtime(&unix_time);
00081     if (!timeinfo) {
00082         uart_print("Failed to convert Unix time", VerboseLevel::ERROR);
00083         return -1;
00084     }
00085
00086     uint8_t time_data[7];
00087     // Convert directly to BCD
00088     time_data[0] = ((timeinfo->tm_sec / 10) << 4) | (timeinfo->tm_sec % 10);
00089     time_data[1] = ((timeinfo->tm_min / 10) << 4) | (timeinfo->tm_min % 10);
00090     time_data[2] = ((timeinfo->tm_hour / 10) << 4) | (timeinfo->tm_hour % 10);
00091     time_data[3] = timeinfo->tm_wday == 0 ? 7 : timeinfo->tm_wday;
00092     time_data[4] = ((timeinfo->tm_mday / 10) << 4) | (timeinfo->tm_mday % 10);
00093     time_data[5] = (((timeinfo->tm_mon + 1) / 10) << 4) | ((timeinfo->tm_mon + 1) % 12);
00094     time_data[6] = (((timeinfo->tm_year - 100) / 10) << 4) | ((timeinfo->tm_year - 100) % 100);
00095

```

```
00096     if (i2c_write_reg(DS3231_SECONDS_REG, 7, time_data) != 0) {
00097         uart_print("Failed to write time to RTC", VerbosityLevel::ERROR);
00098     }
00099     return -1;
00100 }
00101
00102     return 0;
00103 }
00104
00105
00118 int DS3231::read_temperature(float *resolution) {
00119     std::string status;
00120     uint8_t temp[2];
00121     int result = i2c_read_reg(DS3231_TEMPERATURE_MSB_REG, 2, temp);
00122     if (result != 0) {
00123         status = "Failed to read temperature from DS3231";
00124         uart_print(status, VerbosityLevel::ERROR);
00125         return -1;
00126     }
00127
00128     int8_t temperature_msb = (int8_t)temp[0];
00129     uint8_t temperature_lsb = temp[1] >> 6; // Only the 2 MSB are valid
00130
00131     *resolution = temperature_msb + (temperature_lsb * 0.25f); // 0.25 degree resolution
00132
00133     return 0;
00134 }
00135
00136
00146 int16_t DS3231::get_timezone_offset() const {
00147     return timezone_offset_minutes_;
00148 }
00149
00150
00162 void DS3231::set_timezone_offset(int16_t offset_minutes) {
00163     // Validate range: -12 hours to +12 hours (-720 to +720 minutes)
00164     if (offset_minutes >= -720 && offset_minutes <= 720) {
00165         timezone_offset_minutes_ = offset_minutes;
00166     } else {
00167         uart_print("Error: Invalid timezone offset", VerbosityLevel::ERROR);
00168     }
00169 }
00170
00179 time_t DS3231::get_local_time() {
00180     time_t utc_time = get_time();
00181     if (utc_time == -1) {
00182         return -1;
00183     }
00184
00185     return utc_time + (timezone_offset_minutes_ * 60);
00186 }
00187
00188
00189 // ===== private methods
00190
00206 int DS3231::i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
00207     if (!length)
00208         return -1;
00209     std::string status;
00210
00211     recursive_mutex_enter_blocking(&clock_mutex_);
00212     uint8_t reg = reg_addr;
00213     int write_result = i2c_write_blocking(i2c, ds3231_addr, &reg, 1, true);
00214     if (write_result == PICO_ERROR_GENERIC) {
00215         status = "Failed to write register address to DS3231";
00216         uart_print(status, VerbosityLevel::ERROR);
00217         recursive_mutex_exit(&clock_mutex_);
00218         return -1;
00219     }
00220     int read_result = i2c_read_blocking(i2c, ds3231_addr, data, length, false);
00221     if (read_result == PICO_ERROR_GENERIC) {
00222         status = "Failed to read register data from DS3231";
00223         uart_print(status, VerbosityLevel::ERROR);
00224         recursive_mutex_exit(&clock_mutex_);
00225         return -1;
00226     }
00227     recursive_mutex_exit(&clock_mutex_);
00228
00229     return 0;
00230 }
00231
00247 int DS3231::i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
00248     if (!length)
00249         return -1;
00250
00251     recursive_mutex_enter_blocking(&clock_mutex_);
00252     std::vector<uint8_t> message(length + 1);
```

```

00253     message[0] = reg_addr;
00254     for (size_t i = 0; i < length; i++) {
00255         message[i + 1] = data[i];
00256     }
00257     int write_result = i2c_write_blocking(i2c, ds3231_addr, message.data(), (length + 1), false);
00258     if (write_result == PICO_ERROR_GENERIC) {
00259         uart_print("Error: i2c_write_blocking failed in i2c_write_reg", VerboseLevel::ERROR);
00260         recursive_mutex_exit(&clock_mutex_);
00261         return -1;
00262     }
00263     recursive_mutex_exit(&clock_mutex_);
00264
00265     return 0;
00266 } // End of DS3231_RTC group

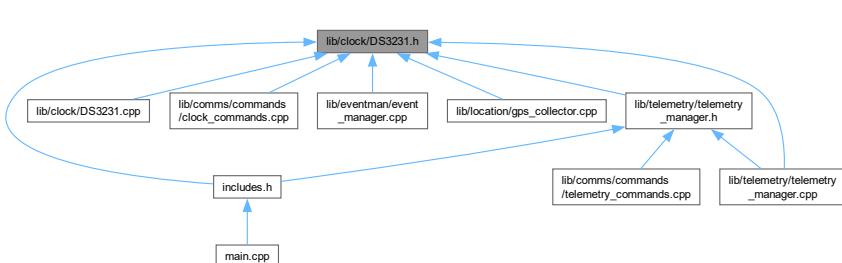
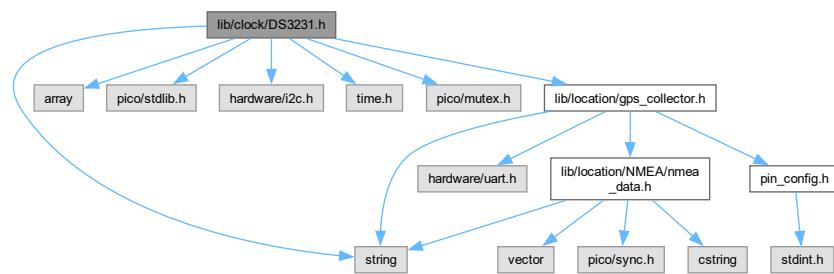
```

9.8 lib/clock/DS3231.h File Reference

```

#include <string>
#include <array>
#include "pico/stl.h"
#include "hardware/i2c.h"
#include <time.h>
#include "pico/mutex.h"
#include "lib/location/gps_collector.h"
Include dependency graph for DS3231.h:

```



Classes

- struct [ds3231_data_t](#)
Structure to hold time and date information from DS3231.
- class [DS3231](#)
Class for interfacing with the DS3231 real-time clock.

Macros

- `#define DS3231_DEVICE_ADDRESS 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.

9.8.1 Macro Definition Documentation

9.8.1.1 DS3231_DEVICE_ADDRESS

```
#define DS3231_DEVICE_ADDRESS 0x68
```

`DS3231` I2C device address.

Definition at line 15 of file `DS3231.h`.

9.8.1.2 DS3231_SECONDS_REG

```
#define DS3231_SECONDS_REG 0x00
```

Register address: Seconds (0-59)

Definition at line 20 of file `DS3231.h`.

9.8.1.3 DS3231_MINUTES_REG

```
#define DS3231_MINUTES_REG 0x01
```

Register address: Minutes (0-59)

Definition at line [25](#) of file [DS3231.h](#).

9.8.1.4 DS3231_HOURS_REG

```
#define DS3231_HOURS_REG 0x02
```

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

Definition at line [30](#) of file [DS3231.h](#).

9.8.1.5 DS3231_DAY_REG

```
#define DS3231_DAY_REG 0x03
```

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

Definition at line [35](#) of file [DS3231.h](#).

9.8.1.6 DS3231_DATE_REG

```
#define DS3231_DATE_REG 0x04
```

Register address: Date (1-31)

Definition at line [40](#) of file [DS3231.h](#).

9.8.1.7 DS3231_MONTH_REG

```
#define DS3231_MONTH_REG 0x05
```

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

Definition at line [45](#) of file [DS3231.h](#).

9.8.1.8 DS3231_YEAR_REG

```
#define DS3231_YEAR_REG 0x06
```

Register address: Year (00-99)

Definition at line [50](#) of file [DS3231.h](#).

9.8.1.9 DS3231_CONTROL_REG

```
#define DS3231_CONTROL_REG 0x0E
```

Register address: Control register.

Definition at line 55 of file [DS3231.h](#).

9.8.1.10 DS3231_CONTROL_STATUS_REG

```
#define DS3231_CONTROL_STATUS_REG 0x0F
```

Register address: Control/Status register.

Definition at line 60 of file [DS3231.h](#).

9.8.1.11 DS3231_TEMPERATURE_MSB_REG

```
#define DS3231_TEMPERATURE_MSB_REG 0x11
```

Register address: Temperature register (MSB)

Definition at line 65 of file [DS3231.h](#).

9.8.1.12 DS3231_TEMPERATURE_LSB_REG

```
#define DS3231_TEMPERATURE_LSB_REG 0x12
```

Register address: Temperature register (LSB)

Definition at line 70 of file [DS3231.h](#).

9.8.2 Enumeration Type Documentation

9.8.2.1 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](#).

9.9 DS3231.h

[Go to the documentation of this file.](#)

```

00001 #ifndef DS3231_H
00002 #define DS3231_H
00003
00004 #include <string>
00005 #include <array>
00006 #include "pico/stdlib.h"
00007 #include "hardware/i2c.h"
00008 #include <time.h>
00009 #include "pico/mutex.h"
00010 #include "lib/location/gps_collector.h"
00011
00015 #define DS3231_DEVICE_ADDRESS 0x68
00016
00020 #define DS3231_SECONDS_REG 0x00
00021
00025 #define DS3231_MINUTES_REG 0x01
00026
00030 #define DS3231_HOURS_REG 0x02
00031
00035 #define DS3231_DAY_REG 0x03
00036
00040 #define DS3231_DATE_REG 0x04
00041
00045 #define DS3231_MONTH_REG 0x05
00046
00050 #define DS3231_YEAR_REG 0x06
00051
00055 #define DS3231_CONTROL_REG 0x0E
00056
00060 #define DS3231_CONTROL_STATUS_REG 0x0F
00061
00065 #define DS3231_TEMPERATURE_MSB_REG 0x11
00066
00070 #define DS3231_TEMPERATURE_LSB_REG 0x12
00071
00076 enum days_of_week {
00077     MONDAY = 1,
00078     TUESDAY,
00079     WEDNESDAY,
00080     THURSDAY,
00081     FRIDAY,
00082     SATURDAY,
00083     SUNDAY
00084 };
00085
00090 typedef struct {
00091     uint8_t seconds;
00092     uint8_t minutes;
00093     uint8_t hours;
00094     uint8_t day;
00095     uint8_t date;
00096     uint8_t month;
00097     uint8_t year;
00098     bool century;
00099 } ds3231_data_t;
00100
00108 class DS3231 {
00109 public:
00115     DS3231(i2c_inst_t *i2c_instance);
00121     static DS3231& get_instance();
00122
00128     int set_time(time_t unix_time);
00129
00134     time_t get_time();
00135
00142     int read_temperature(float *resolution);
00143
00144
00150     int16_t get_timezone_offset() const;
00151
00157     void set_timezone_offset(int16_t offset_minutes);
00158
00164     time_t get_local_time();
00165
00166
00167
00168 private:
00169     i2c_inst_t *i2c;
00170     uint8_t ds3231_addr;
00171     recursive_mutex_t clock_mutex_;
00172     int16_t timezone_offset_minutes_ = 60;
00173     uint32_t sync_interval_minutes_ = 1440;

```

```

00174     time_t last_sync_time_ = 0;
00175
00176     // Private constructor
00177     DS3231();
00178
00179     // Delete copy constructor and assignment operator
00180     DS3231(const DS3231&) = delete;
00181     DS3231& operator=(const DS3231&) = delete;
00182
00183     int i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data);
00184
00185     int i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data);
00186
00187 };
00188
00189 #endif // DS3231_H

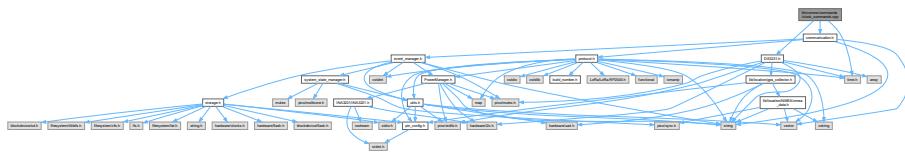
```

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

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

```
#include "DS3231.h"
```

Include dependency graph for clock_commands.cpp:



Functions

- std::vector< Frame > handle_time (const std::string ¶m, OperationType operationType)
Handler for getting and setting system time.
- std::vector< Frame > handle_timezone_offset (const std::string ¶m, OperationType operationType)
Handler for getting and setting timezone offset.
- std::vector< Frame > handle_get_internal_temperature (const std::string ¶m, OperationType operationType)
Handler for reading the DS3231's internal temperature sensor.

Variables

- static constexpr uint8_t clock_commands_group_id = 3
- static constexpr uint8_t time_command_id = 0
- static constexpr uint8_t timezone_offset_command_id = 1
- static constexpr uint8_t internal_temperature_command_id = 4

9.10.1 Variable Documentation

9.10.1.1 clock_commands_group_id

```
uint8_t clock_commands_group_id = 3 [static], [constexpr]
```

Definition at line 5 of file [clock_commands.cpp](#).

9.10.1.2 time_command_id

```
uint8_t time_command_id = 0 [static], [constexpr]
```

Definition at line 6 of file [clock_commands.cpp](#).

9.10.1.3 timezone_offset_command_id

```
uint8_t timezone_offset_command_id = 1 [static], [constexpr]
```

Definition at line 7 of file [clock_commands.cpp](#).

9.10.1.4 internal_temperature_command_id

```
uint8_t internal_temperature_command_id = 4 [static], [constexpr]
```

Definition at line 8 of file [clock_commands.cpp](#).

9.11 clock_commands.cpp

[Go to the documentation of this file.](#)

```
00001 #include "communication.h"
00002 #include <time.h>
00003 #include "DS3231.h" // Include the DS3231 header
00004
00005 static constexpr uint8_t clock_commands_group_id = 3;
00006 static constexpr uint8_t time_command_id = 0;
00007 static constexpr uint8_t timezone_offset_command_id = 1;
00008 static constexpr uint8_t internal_temperature_command_id = 4;
00014
00015
00029 std::vector<Frame> handle_time(const std::string& param, OperationType operationType) {
00030     std::vector<Frame> frames;
00031     std::string error_msg;
00032
00033     if (operationType == OperationType::SET) {
00034         if (param.empty()) {
00035             error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00036             frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id, time_command_id,
00037                 error_msg));
00038             return frames;
00039         }
00039         try {
00040             time_t newTime = std::stoll(param);
00041             if (newTime <= 1742487032 || newTime >= 1893520044) {
00042                 error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00043                 frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00043                     time_command_id, error_msg));
00044                 return frames;
00045             }
00046
00047             if (DS3231::get_instance().set_time(newTime) != 0) {
00048                 error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
00049                 frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00049                     time_command_id, error_msg));
00050                 return frames;
00051             }
00052
00053             EventEmitter::emit(EventGroup::CLOCK, ClockEvent::CHANGED);
00054             frames.push_back(frame_build(OperationType::RES, clock_commands_group_id, time_command_id,
00054                 std::to_string(DS3231::get_instance().get_time())));
00055             return frames;
00056         } catch (...) {
00057             error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
00058             frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id, time_command_id,
00058                 error_msg));
00059         }
00059     }
00059 }
```

```

00060         }
00061     } else if (operationType == OperationType::GET) {
00062         if (!param.empty()) {
00063             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00064             frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id, time_command_id,
00065                                     error_msg));
00066             return frames;
00067         }
00068         uint32_t time_unix = DS3231::get_instance().get_local_time();
00069         if (time_unix == 0) {
00070             error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
00071             frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id, time_command_id,
00072                                     error_msg));
00072             return frames;
00073         }
00074         frames.push_back(frame_build(OperationType::VAL, clock_commands_group_id, time_command_id,
00075                                     std::to_string(time_unix)));
00076         return frames;
00077     }
00078     error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00079     frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id, time_command_id,
00080                             error_msg));
00081     return frames;
00082 }
00083
00094 std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType) {
00095     std::vector<Frame> frames;
00096     std::string error_msg;
00097
00098     if (!(operationType == OperationType::GET || operationType == OperationType::SET)) {
00099         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00100         frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00101                                     timezone_offset_command_id, error_msg));
00101         return frames;
00102     }
00103
00104     if (operationType == OperationType::GET) {
00105         if (!param.empty()) {
00106             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00107             frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00108                                     timezone_offset_command_id, error_msg));
00109             return frames;
00110         }
00111         int offset = DS3231::get_instance().get_timezone_offset();
00112         std::string offset_set = std::to_string(offset);
00113         frames.push_back(frame_build(OperationType::VAL, clock_commands_group_id,
00114                                     timezone_offset_command_id, offset_set));
00115     }
00116
00117     if (param.empty()) {
00118         error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00119         frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00120                                     timezone_offset_command_id, error_msg));
00121         return frames;
00122     }
00123
00124     try {
00125         int16_t offset = std::stoi(param);
00126         if (offset < -720 || offset > 720) {
00127             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00128             frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00129                                     timezone_offset_command_id, error_msg));
00130             return frames;
00131         }
00132         DS3231::get_instance().set_timezone_offset(offset);
00133         std::string offset_set = std::to_string(offset);
00134         frames.push_back(frame_build(OperationType::RES, clock_commands_group_id,
00135                                     timezone_offset_command_id, offset_set));
00136     } catch (...) {
00137         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00138         frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00139                                     timezone_offset_command_id, error_msg));
00140     }
00141
00153 std::vector<Frame> handle_get_internal_temperature(const std::string& param, OperationType
00154 operationType) {
00155     std::vector<Frame> frames;

```

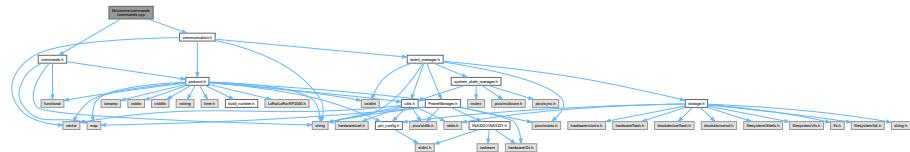
```

00155     std::string error_msg;
00156
00157     if (operationType != OperationType::GET || !param.empty()) {
00158         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00159         frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00160             internal_temperature_command_id, error_msg));
00160         return frames;
00161     }
00162
00163     float temperature;
00164     if (DS3231::get_instance().read_temperature(&temperature) != 0) {
00165         error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
00166         frames.push_back(frame_build(OperationType::ERR, clock_commands_group_id,
00167             internal_temperature_command_id, error_msg));
00167         return frames;
00168     }
00169
00170     std::stringstream ss;
00171     ss << std::fixed << std::precision(2) << temperature;
00172     frames.push_back(frame_build(OperationType::VAL, clock_commands_group_id,
00173         internal_temperature_command_id, ss.str(), ValueUnit::CELSIUS));
00174
00175 }
00176 // end of ClockCommands group

```

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

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



Macros

- #define CMD(group, cmd)

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 ¶m, OperationType operationType)
Executes a command based on its key.

Variables

- `CommandMap command_handlers`

Global map of all command handlers.

9.13 commands.cpp

Go to the documentation of this file.

```

00001 // commands/commands.cpp
00002 #include "commands.h"
00003 #include "communication.h"
00004
00010 #define CMD(group, cmd) ((static_cast<uint32_t>(group) << 8) | static_cast<uint32_t>(cmd))
00011
00016 using CommandHandler = std::function<std::vector<Frame>(<const std::string&, OperationType)>;
00017
00022 using CommandMap = std::map<uint32_t, CommandHandler>;
00023
00028 CommandMap command_handlers = {
00029     {CMD(1, 0), handle_get_commands_list},           // Group 1, Command 0
00030     {CMD(1, 1), handle_get_build_version},          // Group 1, Command 1
00031     {CMD(1, 2), handle_get_power_mode},            // Group 1, Command 2
00032     {CMD(1, 3), handle_get_uptime},                 // Group 1, Command 3
00033     {CMD(1, 8), handle_verbosity},                  // Group 1, Command 8
00034     {CMD(1, 9), handle_enter_bootloader_mode},      // Group 1, Command 9
00035
00036     {CMD(3, 0), handle_time},                      // Group 3, Command 0
00037     {CMD(3, 1), handle_timezone_offset},           // Group 3, Command 1
00038     {CMD(3, 4), handle_get_internal_temperature},   // Group 3, Command 4
00039
00040     {CMD(5, 1), handle_get_last_events},           // Group 5, Command 1
00041     {CMD(5, 2), handle_get_event_count},           // Group 5, Command 2
00042
00043     {CMD(7, 1), handle_gps_power_status},          // Group 7, Command 1
00044     {CMD(7, 2), handle_enable_gps_uart_passthrough}, // Group 7, Command 2
00045
00046     {CMD(8, 2), handle_get_last_telemetry_record}, // Group 8, Command 2
00047     {CMD(8, 3), handle_get_last_sensor_record},     // Group 8, Command 3
00048 };
00049
00050
00059 std::vector<Frame> execute_command(uint32_t commandKey, <const std::string& param, OperationType
operationType) {
00060     auto it = command_handlers.find(commandKey);
00061     if (it != command_handlers.end()) {
00062         CommandHandler handler = it->second;
00063         return handler(param, operationType);
00064     } else {
00065         std::vector<Frame> frames;
00066         frames.push_back(frame_build(OperationType::ERR, 0, 0, "INVALID COMMAND"));
00067         return frames;
00068     }
00069 } // end of CommandSystem group

```

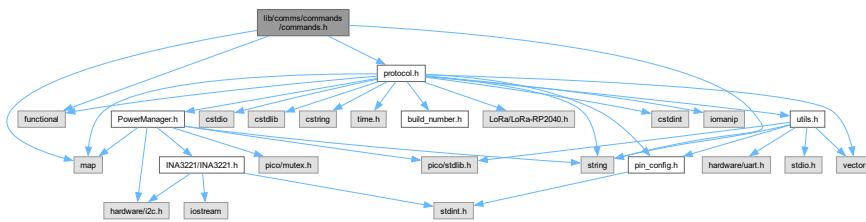
9.14 lib/comms/commands/commands.h File Reference

```

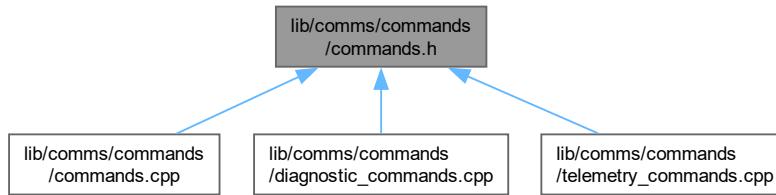
#include <string>
#include <functional>
#include <map>
#include "protocol.h"

```

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 ¶m, OperationType operationType)

Handler for getting and setting system time.
- std::vector< Frame > handle_timezone_offset (const std::string ¶m, OperationType operationType)

Handler for getting and setting timezone offset.
- std::vector< Frame > handle_get_internal_temperature (const std::string ¶m, OperationType operationType)

Handler for reading the DS3231's internal temperature sensor.
- std::vector< Frame > handle_get_commands_list (const std::string ¶m, OperationType operationType)

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

Get firmware build version.
- std::vector< Frame > handle_get_power_mode (const std::string ¶m, OperationType operationType)

Get system power mode.
- std::vector< Frame > handle_get_uptime (const std::string ¶m, OperationType operationType)

Get system uptime.
- std::vector< Frame > handle_verbosity (const std::string ¶m, OperationType operationType)

Handles setting or getting the UART verbosity level.
- std::vector< Frame > handle_enter_bootloader_mode (const std::string ¶m, OperationType operationType)

Reboot system to USB firmware loader.
- std::vector< Frame > handle_gps_power_status (const std::string ¶m, OperationType operationType)

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

- Handler for enabling GPS transparent mode (UART pass-through)*
- std::vector< Frame > handle_get_last_events (const std::string ¶m, OperationType operationType)

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

Handler for getting total number of events in the log.
 - std::vector< Frame > handle_get_last_telemetry_record (const std::string ¶m, OperationType operationType)

Handles the get last record command.
 - std::vector< Frame > handle_get_last_sensor_record (const std::string ¶m, OperationType operationType)

Handles the get last sensor record command.
 - std::vector< Frame > execute_command (uint32_t commandKey, const std::string ¶m, 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.

9.15 commands.h

[Go to the documentation of this file.](#)

```

00001 // commands/commands.h
00002 #ifndef COMMANDS_H
00003 #define COMMANDS_H
00004
00005 #include <string>
00006 #include <functional>
00007 #include <map>
00008 #include "protocol.h"
00009
00010 // CLOCK
00011 std::vector<Frame> handle_time(const std::string& param, OperationType operationType);
00012 std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType);
00013 std::vector<Frame> handle_get_internal_temperature(const std::string& param, OperationType
00014 operationType);
00015
00016 // DIAG
00017 std::vector<Frame> handle_get_commands_list(const std::string& param, OperationType operationType);
00018 std::vector<Frame> handle_get_build_version(const std::string& param, OperationType operationType);
00019 std::vector<Frame> handle_get_power_mode(const std::string& param, OperationType operationType);
00020 std::vector<Frame> handle_get_uptime(const std::string& param, OperationType operationType);
00021 std::vector<Frame> handle_verbosity(const std::string& param, OperationType operationType);
00022 std::vector<Frame> handle_enter_bootloader_mode(const std::string& param, OperationType
00023 operationType);
00024
00025 // GPS
00026 std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType);
00027 std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType
00028 operationType);
00029
00030 // EVENT
00031 std::vector<Frame> handle_get_last_events(const std::string& param, OperationType operationType);
00032 std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType);
00033
00034 // TELEMETRY
00035 std::vector<Frame> handle_get_last_telemetry_record(const std::string& param, OperationType
00036 operationType);
00037 std::vector<Frame> handle_get_last_sensor_record(const std::string& param, OperationType
00038 operationType);

```

```

00039 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
00040 operationType);
00041 extern std::map<uint32_t, std::function<std::vector<Frame>(const std::string&, OperationType)>>
00042 command_handlers;
00043 #endif

```

9.16 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 ¶m, OperationType operationType)
Handler for listing all available commands on UART.
- std::vector< Frame > **handle_get_build_version** (const std::string ¶m, OperationType operationType)
Get firmware build version.
- std::vector< Frame > **handle_get_uptime** (const std::string ¶m, OperationType operationType)
Get system uptime.
- std::vector< Frame > **handle_get_power_mode** (const std::string ¶m, OperationType operationType)
Get system power mode.
- std::vector< Frame > **handle_set_verbosity** (const std::string ¶m, OperationType operationType)
Handles setting or getting the UART verbosity level.
- std::vector< Frame > **handle_enter_bootloader_mode** (const std::string ¶m, OperationType operationType)
Reboot system to USB firmware loader.

Variables

- static constexpr uint8_t **diagnostic_commands_group_id** = 1
- static constexpr uint8_t **commands_list_command_id** = 0
- static constexpr uint8_t **build_version_command_id** = 1
- static constexpr uint8_t **power_mode_command_id** = 2
- static constexpr uint8_t **uptime_command_id** = 3
- static constexpr uint8_t **verbosity_command_id** = 8
- static constexpr uint8_t **enter_bootloader_command_id** = 9

9.17 diagnostic_commands.cpp

Go to the documentation of this file.

```
00001 #include "communication.h"
00002 #include "commands.h"
00003 #include "pico/stdlib.h"
00004 #include "pico/bootrom.h"
00005 #include "system_state_manager.h"
00010
00011 static constexpr uint8_t diagnostic_commands_group_id = 1;
00012 static constexpr uint8_t commands_list_command_id = 0;
00013 static constexpr uint8_t build_version_command_id = 1;
00014 static constexpr uint8_t power_mode_command_id = 2;
00015 static constexpr uint8_t uptime_command_id = 3;
00016 static constexpr uint8_t verbosity_command_id = 8;
00017 static constexpr uint8_t enter_bootloader_command_id = 9;
00018
00019 std::vector<Frame> handle_get_commands_list(const std::string& param, OperationType operationType) {
00020     std::vector<Frame> frames;
00021     std::string error_msg;
00022
00023     if (!(operationType == OperationType::GET)) {
00024         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00025         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00026             commands_list_command_id, error_msg));
00027         return frames;
00028     }
00029
00030     if (!param.empty()) {
00031         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00032         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00033             commands_list_command_id, error_msg));
00034         return frames;
00035     }
00036
00037     std::string combined_command_details;
00038     for (const auto& entry : command_handlers) {
00039         uint32_t command_key = entry.first;
00040         uint8_t group = (command_key >> 8) & 0xFF;
00041         uint8_t command = command_key & 0xFF;
00042
00043         std::string command_details = std::to_string(group) + "." + std::to_string(command);
00044
00045         if (combined_command_details.length() + command_details.length() + 1 > 100) {
00046             frames.push_back(frame_build(OperationType::SEQ, diagnostic_commands_group_id,
00047                 commands_list_command_id, combined_command_details));
00048             combined_command_details = "";
00049         }
00050
00051         if (!combined_command_details.empty()) {
00052             combined_command_details += "-";
00053         }
00054         combined_command_details += command_details;
00055     }
00056
00057     if (!combined_command_details.empty()) {
00058         frames.push_back(frame_build(OperationType::SEQ, diagnostic_commands_group_id,
00059             commands_list_command_id, combined_command_details));
00060     }
00061
00062     frames.push_back(frame_build(OperationType::VAL, diagnostic_commands_group_id,
00063         commands_list_command_id, "SEQ_DONE"));
00064     return frames;
00065 }
00066
00067
00068 std::vector<Frame> handle_get_build_version(const std::string& param, OperationType operationType) {
00069     std::vector<Frame> frames;
00070     std::string error_msg;
00071
00072     if (!(operationType == OperationType::GET)) {
00073         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00074         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00075             build_version_command_id, error_msg));
00076         return frames;
00077     }
00078
00079     if (!param.empty()) {
00080         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00081         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00082             build_version_command_id, error_msg));
00083         return frames;
00084     }
00085 }
```

```

00099     frames.push_back(frame_build(OperationType::VAL, diagnostic_commands_group_id,
00100         build_version_command_id, std::to_string(BUILD_NUMBER)));
00101     return frames;
00102 }
00103
00114 std::vector<Frame> handle_get_uptime(const std::string& param, OperationType operationType) {
00115     std::vector<Frame> frames;
00116     std::string error_msg;
00117
00118     if (!(operationType == OperationType::GET)) {
00119         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00120         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00121             uptime_command_id, error_msg));
00122         return frames;
00123     }
00124
00125     if (!param.empty()) {
00126         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00127         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00128             uptime_command_id, error_msg));
00129         return frames;
00130     }
00131     uint32_t uptime = to_ms_since_boot(get_absolute_time()) / 1000;
00132     frames.push_back(frame_build(OperationType::VAL, diagnostic_commands_group_id, uptime_command_id,
00133         std::to_string(uptime)));
00134     return frames;
00135 }
00136
00146 std::vector<Frame> handle_get_power_mode(const std::string& param, OperationType operationType) {
00147     std::vector<Frame> frames;
00148     std::string error_msg;
00149
00150     if (!(operationType == OperationType::GET)) {
00151         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00152         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00153             power_mode_command_id, error_msg));
00154         return frames;
00155     }
00156
00157     if (!param.empty()) {
00158         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00159         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00160             power_mode_command_id, error_msg));
00161         return frames;
00162     }
00163     SystemOperatingMode mode = SystemStateManager::get_instance().get_operating_mode();
00164     std::string mode_str = (mode == SystemOperatingMode::BATTERY_POWERED) ? "BATTERY" : "USB";
00165     frames.push_back(frame_build(OperationType::VAL, diagnostic_commands_group_id,
00166         power_mode_command_id, mode_str));
00167     return frames;
00168 }
00169
00190 std::vector<Frame> handle_verbosity(const std::string& param, OperationType operationType) {
00191     std::vector<Frame> frames;
00192     std::string error_msg;
00193
00194     if (operationType == OperationType::GET) {
00195         if (!param.empty()) {
00196             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00197             frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00198                 verbosity_command_id, error_msg));
00199             return frames;
00200         }
00201         VerbosityLevel current_level = SystemStateManager::get_instance().get_uart_verbosity();
00202         uart_print("GET_VERBOSITY_" + std::to_string(static_cast<int>(current_level)),
00203                     VerbosityLevel::INFO);
00204         frames.push_back(frame_build(OperationType::VAL, diagnostic_commands_group_id,
00205             verbosity_command_id,
00206                 std::to_string(static_cast<int>(current_level))));
00207         return frames;
00208     }
00209     else if (operationType == OperationType::SET) {
00210         try {
00211             int level = std::stoi(param);
00212             if (level < 0 || level > 4) {
00213                 error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00214                 frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00215                     verbosity_command_id, error_msg));
00216                 return frames;
00217             }
00218             SystemStateManager::get_instance().set_uart_verbosity(static_cast<VerbosityLevel>(level));
00219         }
00220     }
00221 }
```

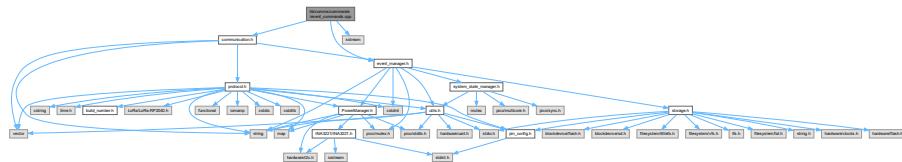
```

00217         uart_print("SET_verbosity_" + std::to_string(level), VerbosityLevel::WARNING);
00218         frames.push_back(frame_build(OperationType::RES, diagnostic_commands_group_id,
00219             verbosity_command_id, "LEVEL SET"));
00220     } catch (...) {
00221         error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
00222         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00223             verbosity_command_id, error_msg));
00224     }
00225 }
00226 else {
00227     error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00228     frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00229         verbosity_command_id, error_msg));
00230 }
00231 }
00232
00243 std::vector<Frame> handle_enter_bootloader_mode(const std::string& param, OperationType operationType)
{
00244     std::vector<Frame> frames;
00245     std::string error_msg;
00246
00247     if (operationType != OperationType::SET) {
00248         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00249         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00250             enter_bootloader_command_id, error_msg));
00251     }
00252
00253     SystemOperatingMode mode = SystemStateManager::get_instance().get_operating_mode();
00254     if (mode == SystemOperatingMode::BATTERY_POWERED) {
00255         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00256         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00257             enter_bootloader_command_id, error_msg));
00258     }
00259
00260     if (param != "USB") {
00261         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00262         frames.push_back(frame_build(OperationType::ERR, diagnostic_commands_group_id,
00263             enter_bootloader_command_id, error_msg));
00264     }
00265
00266     frames.push_back(frame_build(OperationType::RES, diagnostic_commands_group_id,
00267         enter_bootloader_command_id, "REBOOT BOOTSEL"));
00268     SystemStateManager::get_instance().set_bootloader_reset_pending(true);
00269
00270     return frames;
00271 }
00272

```

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

```
#include "communication.h"
#include "event_manager.h"
#include <iostream>
Include dependency graph for event_commands.cpp:
```



Functions

- std::vector< Frame > handle_get_last_events (const std::string ¶m, OperationType operationType)

- Handler for retrieving last N events from the event log.*
- std::vector< Frame > handle_get_event_count (const std::string ¶m, OperationType operationType)
- Handler for getting total number of events in the log.*

Variables

- static constexpr uint8_t event_commands_group_id = 5
- static constexpr uint8_t last_events_command_id = 1
- static constexpr uint8_t event_count_command_id = 2

9.19 event_commands.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002 #include "event_manager.h"
00003 #include <iostream>
00004
00005
00011
00012 static constexpr uint8_t event_commands_group_id = 5;
00013 static constexpr uint8_t last_events_command_id = 1;
00014 static constexpr uint8_t event_count_command_id = 2;
00015
00016
00038 std::vector<Frame> handle_get_last_events(const std::string& param, OperationType operationType) {
00039     std::vector<Frame> frames;
00040     std::string error_msg;
00041
00042     if (operationType != OperationType::GET) {
00043         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00044         frames.push_back(frame_build(OperationType::ERR, event_commands_group_id,
00045             last_events_command_id, error_msg));
00046         return frames;
00047     }
00048     size_t count = 10; // Default number of events to return
00049     if (!param.empty()) {
00050         try {
00051             count = std::stoul(param);
00052             if (count > EVENT_BUFFER_SIZE) {
00053                 error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00054                 frames.push_back(frame_build(OperationType::ERR, event_commands_group_id,
00055                     last_events_command_id, error_msg));
00056             }
00057         } catch (...) {
00058             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00059             frames.push_back(frame_build(OperationType::ERR, event_commands_group_id,
00060                 last_events_command_id, error_msg));
00061             return frames;
00062         }
00063     }
00064     auto& event_manager = EventManager::get_instance();
00065     size_t available = event_manager.get_event_count();
00066     size_t to_return = (count == 0) ? available : std::min(count, available);
00067     size_t event_index = available;
00068
00069     while (to_return > 0) {
00070         std::stringstream ss;
00071         ss << std::hex << std::uppercase << std::setfill('0');
00072         size_t events_in_frame = 0;
00073
00074         for (size_t i = 0; i < 10 && to_return > 0; ++i) {
00075             event_index--;
00076             const EventLog& event = event_manager.get_event(event_index);
00077
00078             ss << std::setw(4) << event.id
00079                 << std::setw(8) << event.timestamp
00080                 << std::setw(2) << static_cast<int>(event.group)
00081                 << std::setw(2) << static_cast<int>(event.event);
00082
00083             if (to_return > 1) ss << "-";
00084             to_return--;
00085         }
00086     }
00087 }
```

```

00085         events_in_frame++;
00086     }
00087     frames.push_back(frame_build(OperationType::SEQ, event_commands_group_id,
00088     last_events_command_id, ss.str()));
00089     frames.push_back(frame_build(OperationType::VAL, event_commands_group_id, last_events_command_id,
00090     "SEQ_DONE"));
00091     return frames;
00092 }
00093
00106 std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType) {
00107     std::vector<Frame> frames;
00108     std::string error_msg;
00109
00110     if (operationType != OperationType::GET || !param.empty()) {
00111         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00112         frames.push_back(frame_build(OperationType::ERR, event_commands_group_id,
00113         event_count_command_id, error_msg));
00114     }
00115
00116     auto& event_manager = EventManager::get_instance();
00117     frames.push_back(frame_build(OperationType::VAL, event_commands_group_id, event_count_command_id,
00118         std::to_string(event_manager.get_event_count())));
00119
00120 } // end of EventCommands group

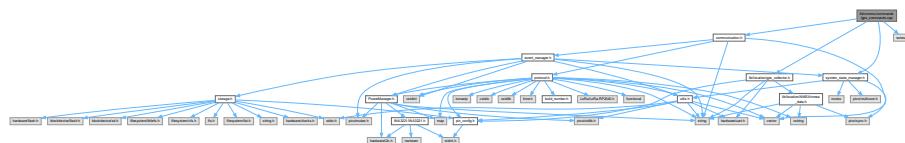
```

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

```

#include "communication.h"
#include "lib/location/gps_collector.h"
#include <iostream>
#include "system_state_manager.h"
Include dependency graph for gps_commands.cpp:

```



Functions

- std::vector< Frame > handle_gps_power_status (const std::string ¶m, OperationType operationType)
Handler for controlling GPS module power state.
- std::vector< Frame > handle_enable_gps_uart_passthrough (const std::string ¶m, OperationType operationType)
Handler for enabling GPS transparent mode (UART pass-through)

Variables

- static constexpr uint8_t gps_commands_group_id = 7
- static constexpr uint8_t power_status_command_id = 1
- static constexpr uint8_t passthrough_command_id = 2

9.20.1 Variable Documentation

9.20.1.1 gps_commands_group_id

```
uint8_t gps_commands_group_id = 7 [static], [constexpr]
```

Definition at line 6 of file [gps_commands.cpp](#).

9.20.1.2 power_status_command_id

```
uint8_t power_status_command_id = 1 [static], [constexpr]
```

Definition at line 7 of file [gps_commands.cpp](#).

9.20.1.3 passthrough_command_id

```
uint8_t passthrough_command_id = 2 [static], [constexpr]
```

Definition at line 8 of file [gps_commands.cpp](#).

9.21 gps_commands.cpp

[Go to the documentation of this file.](#)

```
00001 #include "communication.h"
00002 #include "lib/location/gps_collector.h"
00003 #include <sstream>
00004 #include "system_state_manager.h"
00005
00006 static constexpr uint8_t gps_commands_group_id = 7;
00007 static constexpr uint8_t power_status_command_id = 1;
00008 static constexpr uint8_t passthrough_command_id = 2;
00009
00010
00011 std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType) {
00012     std::vector<Frame> frames;
00013     std::string error_str;
00014
00015     if (operationType == OperationType::SET) {
00016         // command allowed only in ground mode
00017         SystemOperatingMode mode = SystemStateManager::get_instance().get_operating_mode();
00018         if (mode == SystemOperatingMode::BATTERY_POWERED) {
00019             error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
00020             frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00021                 power_status_command_id, error_str));
00022             return frames;
00023         }
00024         if (param.empty()) {
00025             error_str = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00026             frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00027                 power_status_command_id, error_str));
00028             return frames;
00029         }
00030         try {
00031             int power_status = std::stoi(param);
00032             if (power_status != 0 && power_status != 1) {
00033                 error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
00034                 frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00035                     power_status_command_id, error_str));
00036                 return frames;
00037             }
00038             gpio_put(GPS_POWER_ENABLE_PIN, power_status);
00039             EventEmitter::emit(EventGroup::GPS, power_status ? GPSEvent::POWER_ON :
00040                             GPSEvent::POWER_OFF);
00041         }
```

```
00059         frames.push_back(frame_build(OperationType::RES, gps_commands_group_id,
00060             power_status_command_id, std::to_string(power_status)));
00061     } catch (...) {
00062         error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
00063         frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00064             power_status_command_id, error_str));
00065         return frames;
00066     }
00067     else if (operationType == OperationType::GET) {
00068         if (!param.empty()) {
00069             error_str = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00070             frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00071                 power_status_command_id, error_str));
00072         }
00073         bool power_status = gpio_get(GPS_POWER_ENABLE_PIN);
00074         frames.push_back(frame_build(OperationType::VAL, gps_commands_group_id,
00075             power_status_command_id, std::to_string(power_status)));
00076         return frames;
00077     }
00078     error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
00079     frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id, power_status_command_id,
00080         error_str));
00081     return frames;
00082 }
00083
00084
00100 std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType
00101     operationType) {
00102     std::vector<Frame> frames;
00103     std::string error_str;
00104     if (!(operationType == OperationType::SET)) {
00105         error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
00106         frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00107             passthrough_command_id, error_str));
00108         return frames;
00109     }
00110     // disable command if in battery mode
00111     SystemOperatingMode mode = SystemStateManager::get_instance().get_operating_mode();
00112     if (mode == SystemOperatingMode::BATTERY_POWERED) {
00113         error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
00114         frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00115             passthrough_command_id, error_str));
00116         return frames;
00117     }
00118     // Parse and validate timeout parameter
00119     uint32_t timeout_ms;
00120     try {
00121         timeout_ms = param.empty() ? 60000u : std::stoul(param) * 1000;
00122     } catch (...) {
00123         error_str = error_code_to_string(ErrorCode::INVALID_VALUE);
00124         frames.push_back(frame_build(OperationType::ERR, gps_commands_group_id,
00125             passthrough_command_id, error_str));
00126     }
00127
00128     // Setup UART parameters and exit sequence
00129     const std::string EXIT_SEQUENCE = "##EXIT##";
00130     std::string input_buffer;
00131     bool exit_requested = false;
00132     SystemStateManager::get_instance().set_gps_collection_paused(true);
00133     sleep_ms(100);
00134
00135     uint32_t original_baud_rate = DEBUG_UART_BAUD_RATE;
00136     uint32_t gps_baud_rate = GPS_UART_BAUD_RATE;
00137     uint32_t start_time = to_ms_since_boot(get_absolute_time());
00138
00139     EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_START);
00140
00141     std::string message = "Entering GPS Serial Pass-Through Mode @" +
00142         std::to_string(gps_baud_rate) + " for " +
00143         std::to_string(timeout_ms/1000) + "s\r\n" +
00144         "Send " + EXIT_SEQUENCE + " to exit";
00145     uart_print(message, VerbosityLevel::INFO);
00146
00147     sleep_ms(10);
00148
00149     // Change main UART baudrate to GPS module baudrate for passthrough duration
00150     uart_set_baudrate(DEBUG_UART_PORT, gps_baud_rate);
00151
```

```

00152     while (!exit_requested) {
00153         while (uart_is_readable(DEBUG_UART_PORT)) {
00154             char ch = uart_getc(DEBUG_UART_PORT);
00155
00156             input_buffer += ch;
00157             if (input_buffer.length() > EXIT_SEQUENCE.length()) {
00158                 input_buffer = input_buffer.substr(1);
00159             }
00160
00161             if (input_buffer == EXIT_SEQUENCE) {
00162                 exit_requested = true;
00163                 break;
00164             }
00165
00166             if (input_buffer != EXIT_SEQUENCE.substr(0, input_buffer.length())) {
00167                 uart_write_blocking(GPS_UART_PORT,
00168                     reinterpret_cast<const uint8_t*>(&ch), 1);
00169             }
00170         }
00171
00172         while (uart_is_readable(GPS_UART_PORT)) {
00173             char gps_byte = uart_getc(GPS_UART_PORT);
00174             uart_write_blocking(DEBUG_UART_PORT,
00175                 reinterpret_cast<const uint8_t*>(&gps_byte), 1);
00176         }
00177
00178         if (to_ms_since_boot(get_absolute_time()) - start_time >= timeout_ms) {
00179             break;
00180         }
00181     }
00182
00183     uart_set_baudrate(DEBUG_UART_PORT, original_baud_rate);
00184
00185     sleep_ms(50);
00186
00187     SystemStateManager::get_instance().set_gps_collection_paused(false);
00188     EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_END);
00189
00190     std::string exit_reason = exit_requested ? "USER_EXIT" : "TIMEOUT";
00191     std::string response = "GPS UART BRIDGE EXIT: " + exit_reason;
00192     uart_print(response, VerbosityLevel::INFO);
00193
00194     frames.push_back(frame_build(OperationType::RES, gps_commands_group_id, passthrough_command_id,
00195         response));
00195
00196 } // end of GPSCommands group

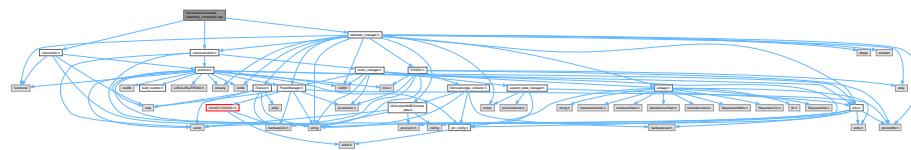
```

9.22 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:

```



Functions

- std::vector< Frame > handle_get_last_telemetry_record (const std::string ¶m, OperationType operationType)

Handles the get last record command.
- std::vector< Frame > handle_get_last_sensor_record (const std::string ¶m, OperationType operationType)

Handles the get last sensor record command.

Variables

- static constexpr uint8_t `telemetry_commands_group` = 8
- static constexpr uint8_t `last_telemetry_command_id` = 2
- static constexpr uint8_t `last_sensor_command_id` = 3

9.22.1 Variable Documentation

9.22.1.1 `telemetry_commands_group`

```
uint8_t telemetry_commands_group = 8 [static], [constexpr]
```

Definition at line 5 of file `telemetry_commands.cpp`.

9.22.1.2 `last_telemetry_command_id`

```
uint8_t last_telemetry_command_id = 2 [static], [constexpr]
```

Definition at line 6 of file `telemetry_commands.cpp`.

9.22.1.3 `last_sensor_command_id`

```
uint8_t last_sensor_command_id = 3 [static], [constexpr]
```

Definition at line 7 of file `telemetry_commands.cpp`.

9.23 `telemetry_commands.cpp`

[Go to the documentation of this file.](#)

```
00001 #include "commands.h"
00002 #include "communication.h"
00003 #include "telemetry_manager.h"
00004
00005 static constexpr uint8_t telemetry_commands_group = 8;
00006 static constexpr uint8_t last_telemetry_command_id = 2;
00007 static constexpr uint8_t last_sensor_command_id = 3;
00008
00009
00010 std::vector<Frame> handle_get_last_telemetry_record([[maybe_unused]] const std::string& param,
00011     OperationType operationType) {
00012     std::vector<Frame> frames;
00013     std::string error_msg;
00014
00015     if (operationType != OperationType::GET) {
00016         if (!param.empty()) {
00017             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00018             frames.push_back(frame_build(OperationType::ERR, telemetry_commands_group,
00019                 last_telemetry_command_id, error_msg));
00020         }
00021     }
00022     error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00023     frames.push_back(frame_build(OperationType::ERR, telemetry_commands_group,
00024         last_telemetry_command_id, error_msg));
00025
00026     return frames;
00027 }
00028
00029 std::string csv_data = TelemetryManager::get_instance().get_last_telemetry_record_csv();
00030 sleep_ms(10);
00031
00032 if (csv_data.empty()) {
```

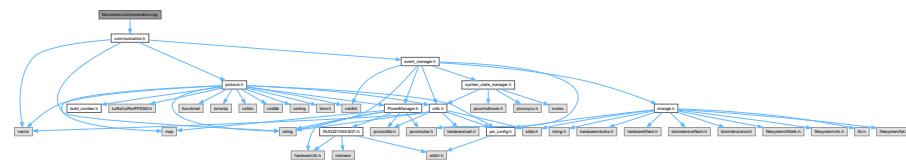
```

00051     error_msg = "NO_DATA";
00052     frames.push_back(frame_build(OperationType::ERR, telemetry_commands_group,
00053         last_sensor_command_id, error_msg));
00054     }
00055     frames.push_back(frame_build(OperationType::VAL, telemetry_commands_group,
00056         last_telemetry_command_id, csv_data));
00057     return frames;
00058 }
00059
00060
00071 std::vector<Frame> handle_get_last_sensor_record([[maybe_unused]] const std::string& param,
00072     OperationType operationType) {
00073     std::vector<Frame> frames;
00074     std::string error_msg;
00075     if (operationType != OperationType::GET) {
00076         if (!(param.empty())) {
00077             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00078             frames.push_back(frame_build(OperationType::ERR, telemetry_commands_group,
00079                 last_telemetry_command_id, error_msg));
00080             return frames;
00081         }
00082         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00083         frames.push_back(frame_build(OperationType::ERR, telemetry_commands_group,
00084             last_sensor_command_id, error_msg));
00085         return frames;
00086     }
00087     std::string csv_data = TelemetryManager::get_instance().get_last_sensor_record_csv();
00088     sleep_ms(10);
00089     if (csv_data.empty()) {
00090         error_msg = "NO_DATA";
00091         frames.push_back(frame_build(OperationType::ERR, telemetry_commands_group,
00092             last_sensor_command_id, error_msg));
00093     }
00094     frames.push_back(frame_build(OperationType::VAL, telemetry_commands_group, last_sensor_command_id,
00095         csv_data));
00096     return frames;
00097 } // TelemetryBufferCommands

```

9.24 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.

9.24.1 Function Documentation

9.24.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

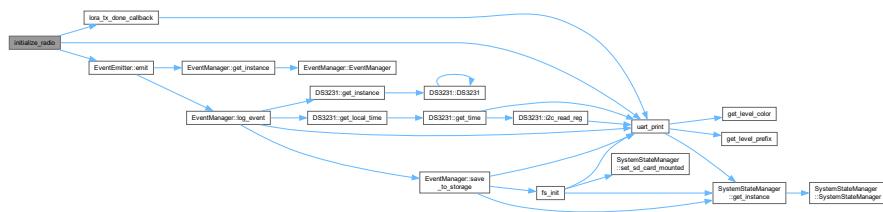
Returns

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 9 of file [communication.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.24.1.2 lora_tx_done_callback()

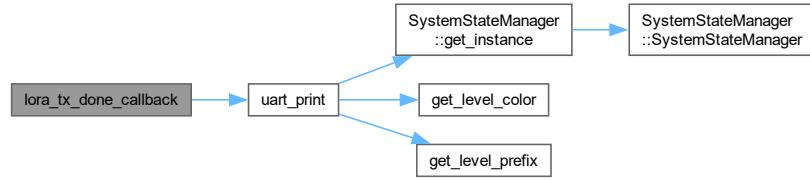
```
void lora_tx_done_callback ()
```

Callback function for LoRa transmission completion.

Prints a debug message to the UART and sets the LoRa module to receive mode.

Definition at line 36 of file [communication.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.25 communication.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00009 bool initialize_radio() {
00010     LoRa.set_pins(lora_cs_pin, lora_reset_pin, lora_irq_pin);
00011     long frequency = 433E6;
00012     bool init_status = false;
00013     if (!LoRa.begin(frequency))
00014     {
00015         uart_print("LoRa init failed. Check your connections.", VerbosityLevel::WARNING);
00016         init_status = false;
00017     } else {
00018         uart_print("LoRa initialized with frequency " + std::to_string(frequency),
00019             VerbosityLevel::INFO);
00020         LoRa.onTxDone(lora_tx_done_callback);
00021
00022         LoRa.receive(0);
00023
00024         init_status = true;
00025     }
00026
00027     EventEmitter::emit(EventGroup::COMMS, init_status ? CommsEvent::RADIO_INIT :
00028         CommsEvent::RADIO_ERROR);
00029
00030     return init_status;
00031 }
00036 void lora_tx_done_callback() {
00037     uart_print("LoRa transmission complete", VerbosityLevel::DEBUG);
00038     LoRa.receive(0);
00039 }
```

9.26 lib/comms/communication.h File Reference

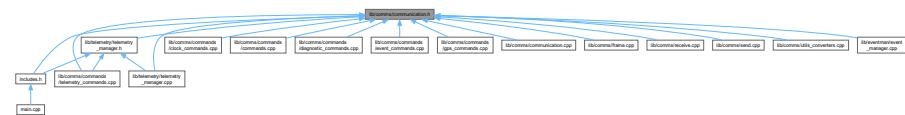
```
#include <string>
#include <vector>
```

```
#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)**
Sends a message using LoRa.
- **void send_frame_uart (const Frame &frame)**
- **void send_frame_lora (const Frame &frame)**
- **std::vector< Frame > execute_command (uint32_t commandKey, const std::string ¶m, 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.

9.26.1 Function Documentation

9.26.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

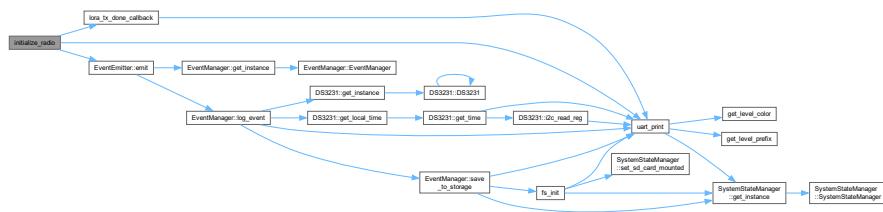
Returns

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 9 of file [communication.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.26.1.2 lora_tx_done_callback()

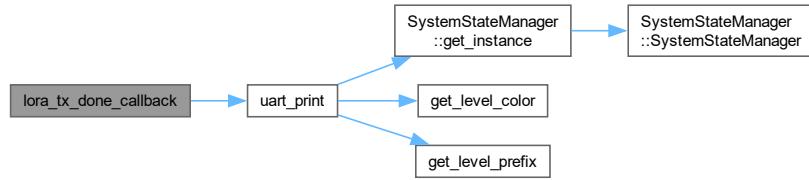
```
void lora_tx_done_callback ()
```

Callback function for LoRa transmission completion.

Prints a debug message to the UART and sets the LoRa module to receive mode.

Definition at line 36 of file [communication.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.26.1.3 on_receive()

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

Callback function for handling received LoRa packets.

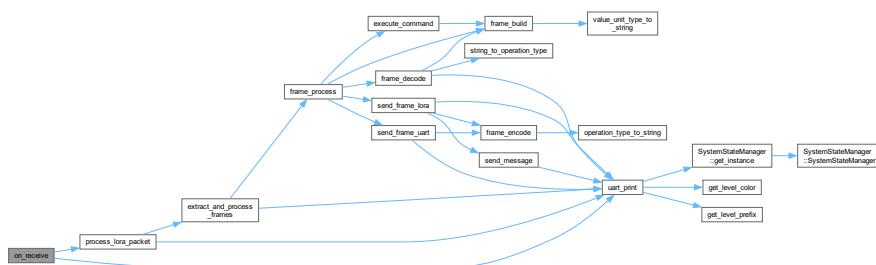
Parameters

<code>packet_size</code>	The size of the received packet.
--------------------------	----------------------------------

Reads the received LoRa packet, extracts metadata, validates addresses, and processes any frames found in the data.

Definition at line 92 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.26.1.4 handle_uart_input()

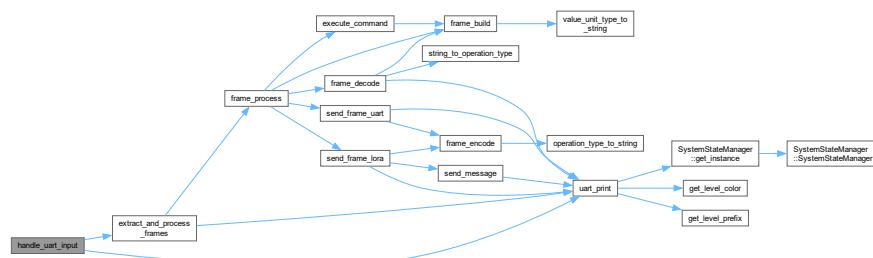
```
void handle_uart_input ()
```

Handles UART input.

Reads characters from the UART port, appends them to a buffer, and processes the buffer when a newline character is received, looking for valid frames in the received data.

Definition at line 121 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.26.1.5 send_message()

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

Sends a message using LoRa.

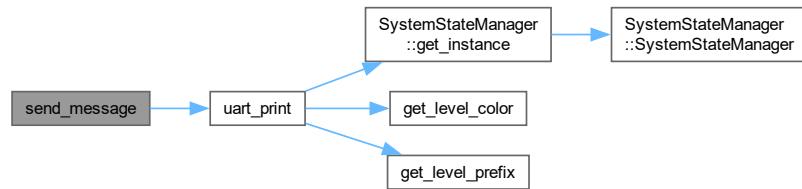
Parameters

<i>outgoing</i>	The message to send.
-----------------	----------------------

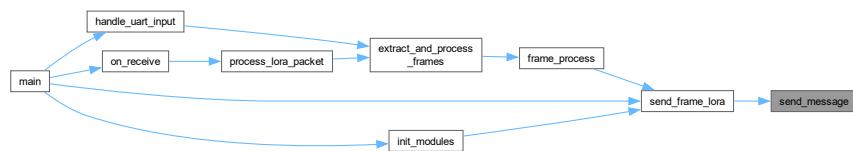
Converts the outgoing string to a C-style string, adds destination and local addresses, and sends the message using LoRa. Prints a log message to the UART.

Definition at line 15 of file [send.cpp](#).

Here is the call graph for this function:



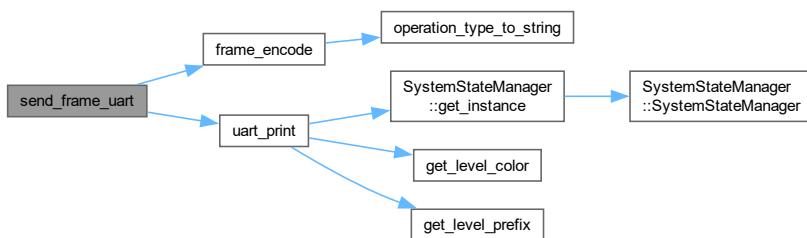
Here is the caller graph for this function:

**9.26.1.6 send_frame_uart()**

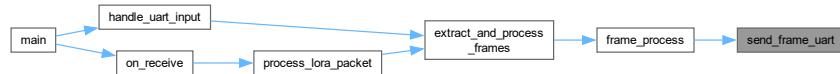
```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 49 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

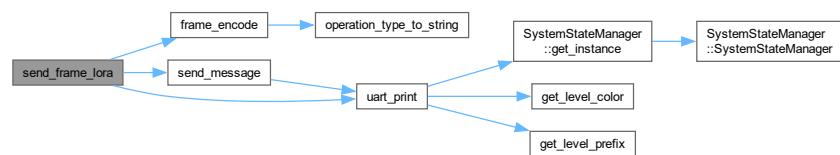


9.26.1.7 send_frame_lora()

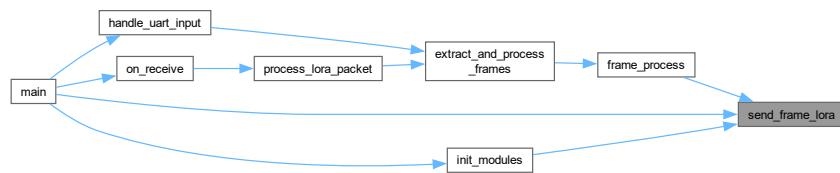
```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 41 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.27 communication.h

[Go to the documentation of this file.](#)

```
00001 #ifndef COMMUNICATION_H
00002 #define COMMUNICATION_H
00003
00004 #include <string>
00005 #include <vector>
00006 #include "protocol.h"
00007 #include "event_manager.h"
00008
00009 bool initialize_radio();
00010 void lora_tx_done_callback();
00011 void on_receive(int packetSize);
00012 void handle_uart_input();
00013 void send_message(std::string outgoing);
```

```

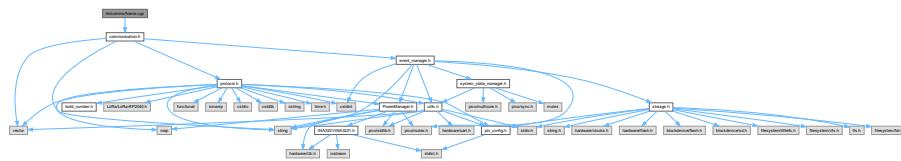
00014 void send_frame_uart(const Frame& frame);
00015 void send_frame_lora(const Frame& frame);
00016
00017 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
00018 operationType);
00019 void frame_process(const std::string& data, Interface interface);
00020 std::string frame_encode(const Frame& frame);
00021 Frame frame_decode(const std::string& data);
00022 Frame frame_build(OperationType operation, uint8_t group, uint8_t command, const std::string& value,
00023 const ValueUnit unitType = ValueUnit::UNDEFINED);
00024 #endif

```

9.28 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.

9.28.1 Detailed Description

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

Definition in file [frame.cpp](#).

9.28.2 Typedef Documentation

9.28.2.1 CommandHandler

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

Definition at line 3 of file frame.cpp.

9.29 frame.cpp

[Go to the documentation of this file.](#)

```
00001 #include "communication.h"
00002
00003 using CommandHandler = std::function<std::vector<Frame>(const std::string&, OperationType)>;
00004 extern std::map<uint32_t, CommandHandler> command_handlers;
00005
00006
00007 std::string frame_encode(const Frame& frame) {
00008     std::stringstream ss;
00009     ss << static_cast<int>(frame.direction) << DELIMITER
00010     << operation_type_to_string(frame.operationType) << DELIMITER
00011     << static_cast<int>(frame.group) << DELIMITER
00012     << static_cast<int>(frame.command) << DELIMITER
00013     << frame.value;
00014
00015     if (!frame.unit.empty()) {
00016         ss << DELIMITER << frame.unit;
00017     }
00018
00019     return FRAME_BEGIN + DELIMITER + ss.str() + DELIMITER + FRAME_END;
00020 }
00021
00022
00023 Frame frame_decode(const std::string& data) {
00024     Frame frame;
00025     frame.header = FRAME_BEGIN;
00026     frame.footer = FRAME_END;
00027     try {
00028         std::stringstream ss(data);
00029         std::string token;
00030
00031         uart_print("Decoding frame: " + data, VerbosityLevel::WARNING);
00032
00033         if (!std::getline(ss, token, DELIMITER) || token != FRAME_BEGIN) {
00034             throw std::runtime_error("DECODE_INVALID_HEADER");
00035         }
00036
00037         std::getline(ss, token, DELIMITER);
00038         int direction = std::stoi(token);
00039         if (direction != 0 && direction != 1) {
00040             throw std::runtime_error("DECODE_INVALID_DIR");
00041         }
00042         frame.direction = direction;
00043
00044         if (!std::getline(ss, token, DELIMITER)) {
00045             throw std::runtime_error("DECODE_MISSING_OP");
00046         }
00047         frame.operationType = string_to_operation_type(token);
00048
00049         if (!std::getline(ss, token, DELIMITER)) {
00050             throw std::runtime_error("DECODE_MISSING_GROUP");
00051         }
00052         int group = std::stoi(token);
00053         if (group < 0 || group > 10) {
00054             throw std::runtime_error("DECODE_INVALID_GROUP");
00055         }
00056         frame.group = group;
00057
00058         if (!std::getline(ss, token, DELIMITER)) {
00059             throw std::runtime_error("DECODE_MISSING_CMD");
00060         }
00061         int command = std::stoi(token);
00062         if (command < 0 || command > 10) {
00063             throw std::runtime_error("DECODE_INVALID_CMD");
00064         }
00065         frame.command = command;
00066     }
00067 }
```

```

00104         if (!std::getline(ss, token, DELIMITER)) frame.value = "";
00105         else frame.value = token;
00106
00107         if (!std::getline(ss, token, DELIMITER)) frame.unit = "";
00108         else frame.unit = token;
00109
00110         std::getline(ss, token, DELIMITER);
00111         if (token != FRAME_END) {
00112             throw std::runtime_error("DECODE_INVALID_FOOTER");
00113         }
00114         frame.footer = token;
00115
00116         return frame;
00117
00118     } catch (const std::exception& e) {
00119         uart_print("Frame decode error: " + std::string(e.what()), VerbosityLevel::ERROR);
00120         return frame_build(OperationType::ERR, 0, 0, e.what());
00121     }
00122 }
00123 }
00124
00125 void frame_process(const std::string& data, Interface interface) {
00126     gpio_put(PICO_DEFAULT_LED_PIN, false);
00127
00128     try {
00129         Frame frame = frame_decode(data);
00130         uint32_t command_key = (static_cast<uint32_t>(frame.group) << 8) |
00131             static_cast<uint32_t>(frame.command);
00132
00133         std::vector<Frame> response_frames = execute_command(command_key, frame.value,
00134             frame.operationType);
00135
00136         gpio_put(PICO_DEFAULT_LED_PIN, true);
00137
00138         // Send all responses through the same interface that received the command
00139         for (const auto& response_frame : response_frames) {
00140             if (interface == Interface::UART) {
00141                 send_frame_uart(response_frame);
00142             } else if (interface == Interface::LORA) {
00143                 send_frame_lora(response_frame);
00144                 sleep_ms(25);
00145             }
00146         }
00147     } catch (const std::exception& e) {
00148         Frame error_frame = frame_build(OperationType::ERR, 0, 0, e.what());
00149         if (interface == Interface::UART) {
00150             send_frame_uart(error_frame);
00151         } else if (interface == Interface::LORA) {
00152             send_frame_lora(error_frame);
00153         }
00154     }
00155 }
00156 }
00157
00158 }
00159 }
00160
00161 Frame frame_build(OperationType operation, uint8_t group, uint8_t command,
00162                     const std::string& value, const ValueUnit unitType) {
00163     Frame frame;
00164     frame.header = FRAME_BEGIN;
00165     frame.footer = FRAME_END;
00166
00167     switch (operation) {
00168         case OperationType::VAL:
00169             frame.direction = 1;
00170             frame.operationType = OperationType::VAL;
00171             frame.value = value;
00172             frame.unit = value_unit_type_to_string(unitType);
00173             break;
00174
00175         case OperationType::ERR:
00176             frame.direction = 1;
00177             frame.operationType = OperationType::ERR;
00178             frame.value = value;
00179             frame.unit = value_unit_type_to_string(ValueUnit::UNDEFINED);
00180             break;
00181
00182         case OperationType::RES:
00183             frame.direction = 1;
00184             frame.operationType = OperationType::RES;
00185             frame.value = value;
00186             frame.unit = value_unit_type_to_string(unitType);
00187             break;
00188
00189         case OperationType::SEQ:
00190             frame.direction = 1;
00191             frame.operationType = OperationType::SEQ;
00192             frame.value = value;
00193             frame.unit = value_unit_type_to_string(unitType);
00194             break;
00195
00196         case OperationType::ACK:
00197             frame.direction = 0;
00198             frame.operationType = OperationType::ACK;
00199             frame.value = value;
00200             frame.unit = value_unit_type_to_string(unitType);
00201             break;
00202
00203     }
00204 }
```

```

00205     default:
00206         break;
00207     }
00208
00209     frame.group = group;
00210     frame.command = command;
00211
00212     return frame;
00213 }
00214 // end of FrameHandling group

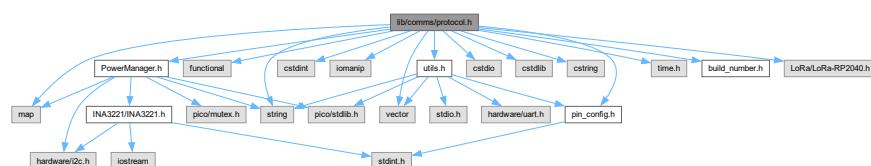
```

9.30 lib/comms/protocol.h File Reference

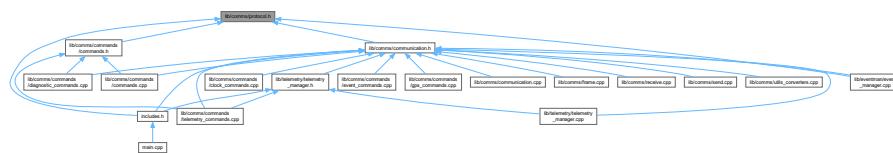
```

#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` : `uint8_t` {
 `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` : `uint8_t` {
 `OperationType::GET` , `OperationType::SET` , `OperationType::RES` , `OperationType::VAL` ,
 `OperationType::SEQ` , `OperationType::ERR` }

Represents the type of operation being performed.
- enum class `ValueUnit` : `uint8_t` {
 `ValueUnit::UNDEFINED` , `ValueUnit::SECOND` , `ValueUnit::VOLT` , `ValueUnit::BOOL` ,
 `ValueUnit::DATETIME` , `ValueUnit::TEXT` , `ValueUnit::MILIAMP` , `ValueUnit::CELSIUS` }

Represents the unit of measurement for a payload value.
- enum class `Interface` : `uint8_t` { `Interface::UART` , `Interface::LORA` }

Represents the communication interface being used.

Functions

- `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::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 = ','`

9.30.1 Variable Documentation

9.30.1.1 FRAME_BEGIN

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

Definition at line 31 of file `protocol.h`.

9.30.1.2 FRAME_END

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

Definition at line 38 of file `protocol.h`.

9.30.1.3 DELIMITER

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

Definition at line 45 of file [protocol.h](#).

9.31 protocol.h

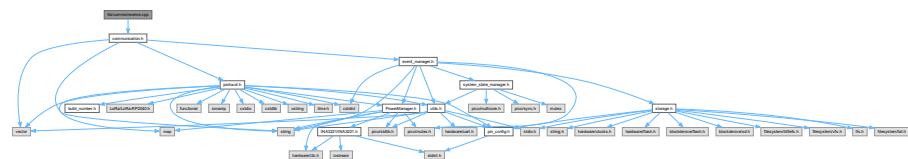
[Go to the documentation of this file.](#)

```
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 : uint8_t {
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 : uint8_t {
00074     GET,
00076     SET,
00078     RES,
00080     VAL,
00082     SEQ,
00084     ERR,
00085
00086 };
00087
00088
00094 enum class ValueUnit : uint8_t {
00096     UNDEFINED,
00098     SECOND,
00100     VOLT,
00102     BOOL,
00104     DATETIME,
00106     TEXT,
00108     MILIAMP,
00110     CELSIUS,
00111 };
00112
00113
00119 enum class Interface : uint8_t {
00121     UART,
00123     LORA
00124 };
```

```
00125
00126
00143 struct Frame {
00144     std::string header;           // Start marker
00145     uint8_t direction;          // 0 = ground->sat, 1 = sat->ground
00146     OperationType operationType;
00147     uint8_t group;              // Group ID
00148     uint8_t command;            // Command ID within group
00149     std::string value;           // Payload value
00150     std::string unit;           // Payload unit
00151     std::string footer;          // End marker
00152 };
00153
00154 std::string error_code_to_string(ErrorCode code);
00155 std::string operation_type_to_string(OperationType type);
00156 OperationType string_to_operation_type(const std::string& str);
00157 std::string value_unit_type_to_string(ValueUnit unit);
00158
00159 #endif
00160
```

9.32 lib/comms/receive.cpp File Reference

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



Macros

- #define MAX_PACKET_SIZE 255

Functions

- bool `extract_and_process_frames` (const std::string &buffer, `Interface` interface)
Extract and process frames from a buffer.
 - bool `process_lora_packet` (const std::vector< uint8_t > &buffer, int bytes_read)
Process LoRa packet metadata and extract frame data.
 - void `on_receive` (int packet_size)
Callback function for handling received LoRa packets.
 - void `handle_uart_input` ()
Handles UART input.

9.32.1 Macro Definition Documentation

9.32.1.1 MAX PACKET SIZE

```
#define MAX_PACKET_SIZE 255
```

Definition at line 3 of file [receive.cpp](#).

9.32.2 Function Documentation

9.32.2.1 extract_and_process_frames()

```
bool extract_and_process_frames (
    const std::string & buffer,
    Interface interface)
```

Extract and process frames from a buffer.

Parameters

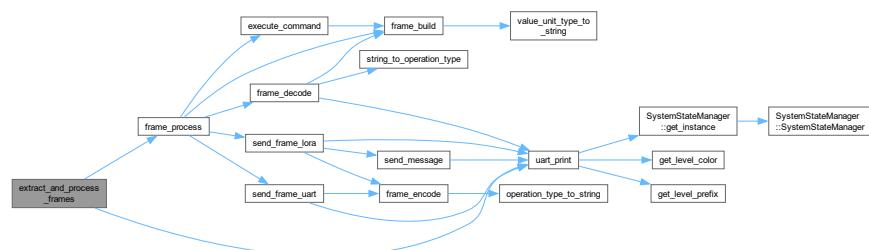
<i>buffer</i>	The buffer containing potential frame data
<i>interface</i>	The interface the data was received on (UART or LoRa)

Returns

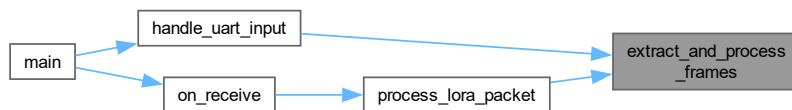
bool True if at least one valid frame was found and processed

Definition at line 12 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.32.2.2 process_lora_packet()

```
bool process_lora_packet (
    const std::vector< uint8_t > & buffer,
    int bytes_read)
```

Process LoRa packet metadata and extract frame data.

Parameters

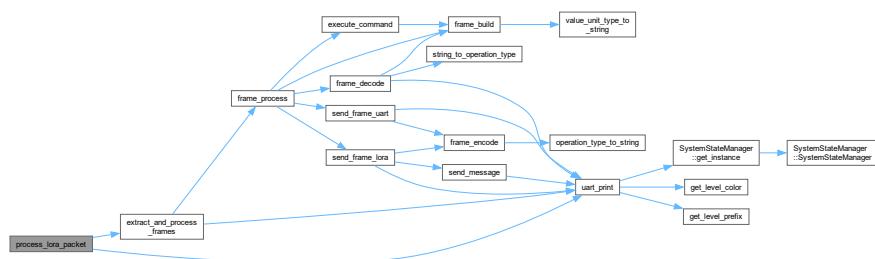
<i>buffer</i>	The raw buffer containing the LoRa packet
<i>bytes_read</i>	The number of bytes in the buffer

Returns

bool True if packet was valid and processed

Definition at line 48 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.32.2.3 on_receive()

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

Callback function for handling received LoRa packets.

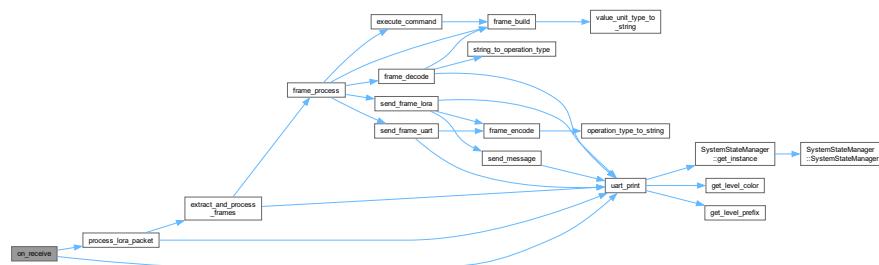
Parameters

<i>packet_size</i>	The size of the received packet.
--------------------	----------------------------------

Reads the received LoRa packet, extracts metadata, validates addresses, and processes any frames found in the data.

Definition at line 92 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.32.2.4 handle_uart_input()

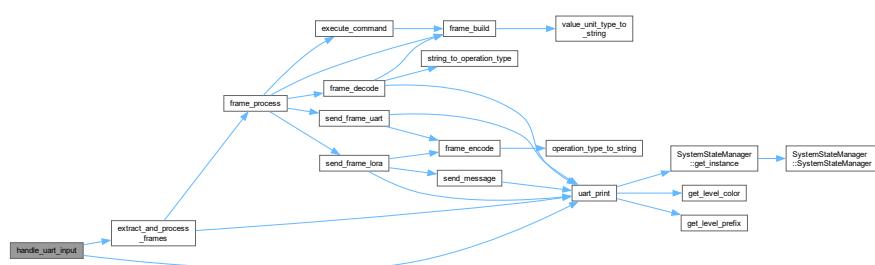
```
void handle_uart_input ()
```

Handles UART input.

Reads characters from the UART port, appends them to a buffer, and processes the buffer when a newline character is received, looking for valid frames in the received data.

Definition at line 121 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.33 receive.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003 #define MAX_PACKET_SIZE 255
00004
00012 bool extract_and_process_frames(const std::string& buffer, Interface interface) {
00013     size_t search_pos = 0;
00014     bool found_frame = false;
00015
00016     while (search_pos < buffer.length()) {
00017         size_t header_pos = buffer.find(FRAME_BEGIN, search_pos);
00018         if (header_pos == std::string::npos) break;
00019
00020         size_t footer_pos = buffer.find(FRAME_END, header_pos);
00021         if (footer_pos == std::string::npos) break;
00022
00023         if (footer_pos > header_pos) {
00024             std::string frame_data = buffer.substr(header_pos, footer_pos + FRAME_END.length() -
00025             header_pos);
00026             uart_print("Extracted frame (length=" + std::to_string(frame_data.length()) +
00027                         "): " + frame_data, VerbosityLevel::DEBUG);
00028             frame_process(frame_data, interface);
00029             found_frame = true;
00030         }
00031         search_pos = footer_pos + FRAME_END.length();
00032     }
00033
00034     if (!found_frame) {
00035         uart_print("No valid frame found in received data", VerbosityLevel::WARNING);
00036     }
00037
00038     return found_frame;
00039 }
00040
00048 bool process_lora_packet(const std::vector<uint8_t>& buffer, int bytes_read) {
00049     if (bytes_read < 2) {
00050         uart_print("Error: Packet too small to contain metadata!", VerbosityLevel::ERROR);
00051         return false;
00052     }
00053
00054     uint8_t received_destination = buffer[0];
00055     uint8_t received_local_address = buffer[1];
00056
00057     if (received_destination != lora_address_local) {
00058         uart_print("Error: Destination address mismatch!", VerbosityLevel::ERROR);
00059         return false;
00060     }
00061
00062     if (received_local_address != lora_address_remote) {
00063         uart_print("Error: Local address mismatch!", VerbosityLevel::ERROR);
00064         return false;
00065     }
00066
00067     // Skip 2 bytes being local and remote address appended by ground station
00068     int start_index = 2;
00069     std::string received(buffer.begin() + start_index, buffer.end());
00070
00071     if (received.empty()) return false;
00072
00073     stringstream hex_dump;

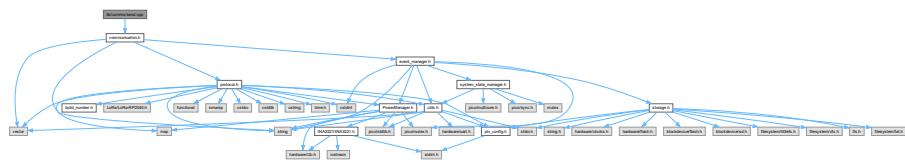
```

```
00074     hex_dump << "Raw bytes: ";
00075     for (int i = 0; i < bytes_read; i++) {
00076         hex_dump << std::hex << std::setfill('0') << std::setw(2)
00077             << static_cast<int>(buffer[i]) << " ";
00078     }
00079     uart_print(hex_dump.str(), VerbosityLevel::DEBUG);
00080
00081     // Extract and process frames using the common function
00082     return extract_and_process_frames(received, Interface::LORA);
00083 }
00084
00092 void on_receive(int packet_size) {
00093     if (packet_size == 0) return;
00094     uart_print("Received LoRa packet of size " + std::to_string(packet_size), VerbosityLevel::DEBUG);
00095
00096     std::vector<uint8_t> buffer;
00097     buffer.reserve(packet_size);
00098
00099     int bytes_read = 0;
00100
00101     while (LoRa.available() && bytes_read < packet_size) {
00102         if (bytes_read >= MAX_PACKET_SIZE) {
00103             uart_print("Error: Packet exceeds maximum allowed size!", VerbosityLevel::ERROR);
00104             return;
00105         }
00106         buffer.push_back(LoRa.read());
00107         bytes_read++;
00108     }
00109
00110     uart_print("Received " + std::to_string(bytes_read) + " bytes", VerbosityLevel::DEBUG);
00111
00112     process_lora_packet(buffer, bytes_read);
00113 }
00114
00121 void handle_uart_input() {
00122     static std::string uart_buffer;
00123
00124     while (uart_is_readable(DEBUG_UART_PORT)) {
00125         char c = uart_getc(DEBUG_UART_PORT);
00126
00127         if (c == '\r' || c == '\n') {
00128             if (!uart_buffer.empty()) {
00129                 uart_print("Received UART string: " + uart_buffer, VerbosityLevel::DEBUG);
00130                 extract_and_process_frames(uart_buffer, Interface::UART);
00131                 uart_buffer.clear();
00132             }
00133         } else {
00134             uart_buffer += c;
00135         }
00136     }
00137 }
```

9.34 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` (std::string outgoing)
Sends a message using LoRa.
 - void `send_frame_lora` (const Frame &frame)
 - void `send_frame_uart` (const Frame &frame)

9.34.1 Detailed Description

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

Definition in file [send.cpp](#).

9.34.2 Function Documentation

9.34.2.1 send_message()

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

Sends a message using LoRa.

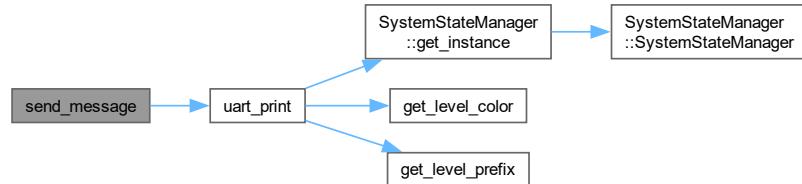
Parameters

<i>outgoing</i>	The message to send.
-----------------	----------------------

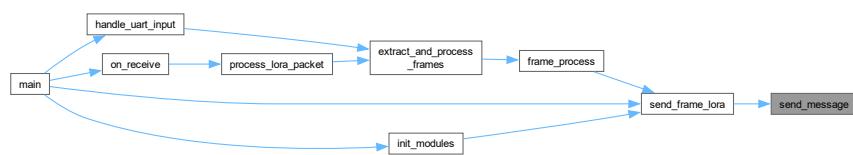
Converts the outgoing string to a C-style string, adds destination and local addresses, and sends the message using LoRa. Prints a log message to the UART.

Definition at line 15 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

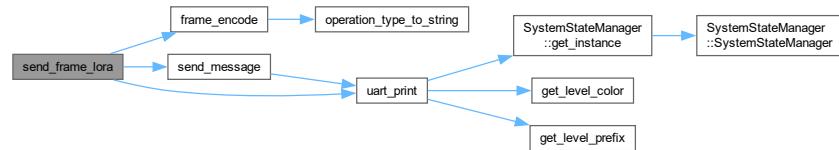


9.34.2.2 send_frame_lora()

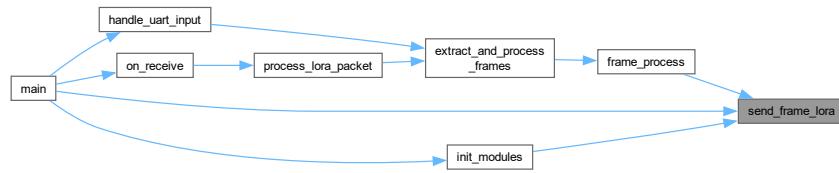
```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 41 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

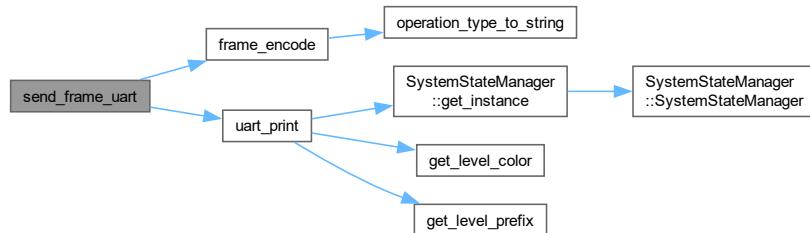


9.34.2.3 send_frame_uart()

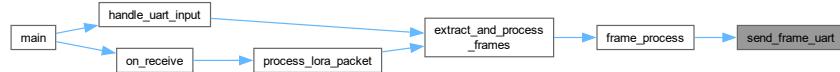
```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 49 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.35 send.cpp

[Go to the documentation of this file.](#)

```

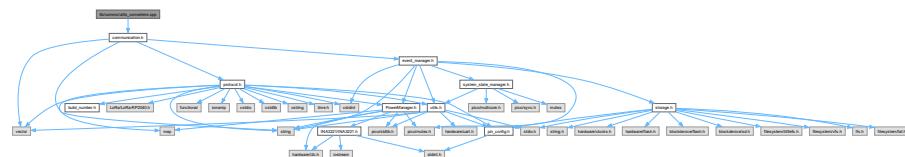
00001 #include "communication.h"
00002
00003
00008
00015 void send_message(std::string outgoing)
00016 {
00017     std::vector<char> send(outgoing.length() + 1);
00018     strcpy(send.data(), outgoing.c_str());
00019
00020     uart_print("LoRa packet begin", VerbosityLevel::DEBUG);
00021     LoRa.beginPacket(); // start packet
00022     LoRa.write(lora_address_remote); // add destination address
00023     LoRa.write(lora_address_local); // add sender address
00024     for(size_t i = 0; i < send.size(); i++) {
00025         LoRa.write(static_cast<uint8_t>(send[i])); // add payload byte by byte
00026     }
00027     LoRa.endPacket(false); // finish packet and send it, param - async
00028
00029     uart_print("LoRa packet end", VerbosityLevel::DEBUG);
00030
00031     std::string message_to_log = "Sent message of size " + std::to_string(send.size());
00032     message_to_log += " to 0x" + std::to_string(lora_address_remote);
00033     message_to_log += " containing: " + std::string(send.data());
00034
00035     uart_print(message_to_log, VerbosityLevel::DEBUG);
00036
00037     LoRa.flush();
00038 }
00039
00040
00041 void send_frame_lora(const Frame& frame) {
00042     uart_print("Sending frame via LoRa", VerbosityLevel::DEBUG);
00043     std::string outgoing = frame_encode(frame);
00044     send_message(outgoing);
00045     uart_print("Frame sent via LoRa", VerbosityLevel::DEBUG);
00046 }
00047
00048 // If level is 0 - SILENT it means no diagnostic output but frame communications should still work
00049 void send_frame_uart(const Frame& frame) {
00050     std::string encoded_frame = frame_encode(frame);
00051     uart_print(encoded_frame, VerbosityLevel::SILENT);
00052 }
00053
00054
  
```

9.36 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 `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.

9.36.1 Detailed Description

Implements utility functions for converting between different data types.

Definition in file `utils_converters.cpp`.

9.37 utils_converters.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003
00009
00010
00017 std::string value_unit_type_to_string(ValueUnit unit) {
00018     switch (unit) {
00019         case ValueUnit::UNDEFINED: return "";
00020         case ValueUnit::SECOND: return "s";
00021         case ValueUnit::VOLT: return "V";
00022         case ValueUnit::BOOL: return "";
00023         case ValueUnit::DATETIME: return "";
00024         case ValueUnit::TEXT: return "";
00025         case ValueUnit::MILIAMP: return "mA";
00026         case ValueUnit::CELSIUS: return "C";
00027         default: return "";
00028     }
00029 }
00030
00031
00038 std::string operation_type_to_string(OperationType type) {
00039     switch (type) {
00040         case OperationType::GET: return "GET";
00041         case OperationType::SET: return "SET";
00042         case OperationType::VAL: return "VAL";
00043         case OperationType::ERR: return "ERR";
00044         case OperationType::RES: return "RES";
00045         case OperationType::SEQ: return "SEQ";
00046         default: return "UNKNOWN";
00047     }
00048 }
00049
00050
00057 OperationType string_to_operation_type(const std::string& str) {
00058     if (str == "GET") return OperationType::GET;
00059     if (str == "SET") return OperationType::SET;
00060     if (str == "VAL") return OperationType::VAL;
00061     if (str == "ERR") return OperationType::ERR;
00062     if (str == "RES") return OperationType::RES;
00063     if (str == "SEQ") return OperationType::SEQ;
00064     return OperationType::GET; // Default to GET
00065 }
00066
00073 std::string error_code_to_string(ErrorCode code) {
00074     switch (code) {
00075         case ErrorCode::PARAM_UNNECESSARY: return "PARAM_UNNECESSARY";
00076         case ErrorCode::PARAM_REQUIRED: return "PARAM_REQUIRED";
00077         case ErrorCode::PARAM_INVALID: return "PARAM_INVALID";
00078         case ErrorCode::INVALID_OPERATION: return "INVALID_OPERATION";

```

```

00079     case ErrorCode::NOT_ALLOWED:           return "NOT_ALLOWED";
00080     case ErrorCode::INVALID_FORMAT:        return "INVALID_FORMAT";
00081     case ErrorCode::INVALID_VALUE:         return "INVALID_VALUE";
00082     case ErrorCode::FAIL_TO_SET:          return "FAIL_TO_SET";
00083     case ErrorCode::INTERNAL_FAIL_TO_READ: return "INTERNAL_FAIL_TO_READ";
00084     default:                            return "UNKNOWN_ERROR";
00085   }
00086 }
00087

```

9.38 lib/eventman/event_manager.cpp File Reference

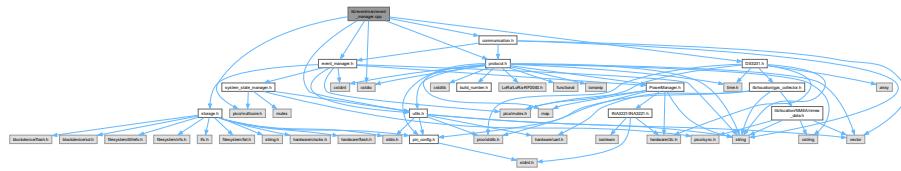
Implementation of the Event Manager and Event Emitter classes.

```

#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:

```



9.38.1 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](#).

9.39 event_manager.cpp

[Go to the documentation of this file.](#)

```

00001
00015
00016 #include "event_manager.h"
00017 #include <cstdio>
00018 #include "protocol.h"
00019 #include "pico/multicore.h"
00020 #include "communication.h"
00021 #include "utils.h"
00022 #include "DS3231.h"
00023
00024
00030 bool EventManager::init() {
00031   if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00032     uart_print("Event manager initialized (storage not available)", VerboseLevel::WARNING);

```

```

00033     return false;
00034 }
00035
00036 FILE* file = fopen(EVENT_LOG_FILE, "w");
00037 if (!file) {
00038     file = fopen(EVENT_LOG_FILE, "w");
00039     if (file) {
00040         fclose(file);
00041         uart_print("Created new event log", VerbosityLevel::INFO);
00042     }
00043     else {
00044         uart_print("Failed to create event log", VerbosityLevel::ERROR);
00045         return false;
00046     }
00047 }
00048 else {
00049     fclose(file);
00050 }
00051
00052 uart_print("Event manager initialized", VerbosityLevel::INFO);
00053 return true;
00054 }
00055
00056
00063 void EventManager::log_event(uint8_t group, uint8_t event) {
00064     mutex_enter_blocking(&eventMutex);
00065
00066     uint32_t timestamp = DS3231::get_instance().get_local_time();
00067     uint16_t id = nextEventId++;
00068
00069     EventLog& log = events[writeIndex];
00070     log.id = id;
00071     log.timestamp = timestamp;
00072     log.group = group;
00073     log.event = event;
00074
00075     writeIndex = (writeIndex + 1) % EVENT_BUFFER_SIZE;
00076     if (eventCount < EVENT_BUFFER_SIZE) {
00077         eventCount++;
00078     }
00079
00080     eventsSinceFlush++;
00081
00082     mutex_exit(&eventMutex);
00083
00084     std::string event_string = "Event: " + std::to_string(id) +
00085         " Group: " + std::to_string(group) +
00086         " Event: " + std::to_string(event);
00087     uart_print(event_string, VerbosityLevel::WARNING);
00088
00089     if (eventsSinceFlush >= EVENT_FLUSH_THRESHOLD || group == static_cast<uint8_t>(EventGroup::POWER))
00090     {
00091         save_to_storage();
00092         eventsSinceFlush = 0;
00093     }
00094
00095
00102 const EventLog& EventManager::get_event(size_t index) const {
00103     mutex_enter_blocking(const_cast<mutex_t*>(&eventMutex));
00104     if (index >= eventCount) {
00105         static EventLog emptyEvent;
00106         mutex_exit(const_cast<mutex_t*>(&eventMutex));
00107         return emptyEvent;
00108     }
00109
00110     size_t readIndex;
00111     if (eventCount == EVENT_BUFFER_SIZE) {
00112         readIndex = (writeIndex + index) % EVENT_BUFFER_SIZE;
00113     }
00114     else {
00115         readIndex = index;
00116     }
00117     const EventLog& event = events[readIndex];
00118     mutex_exit(const_cast<mutex_t*>(&eventMutex));
00119     return event;
00120 }
00121
00122
00128 bool EventManager::save_to_storage() {
00129     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00130         bool status = fs_init();
00131         if (!status) {
00132             return false;
00133         }
00134     }
00135

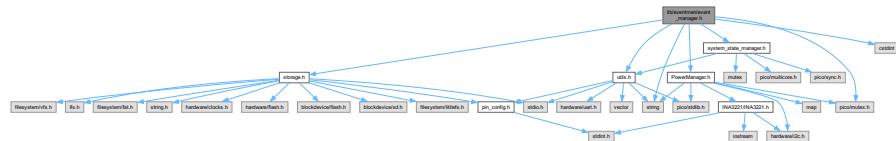
```

```
00136     FILE* file = fopen(EVENT_LOG_FILE, "a");
00137     if (file) {
00138         size_t startIdx = (writeIndex >= eventsSinceFlush) ?
00139             writeIndex - eventsSinceFlush :
00140             EVENT_BUFFER_SIZE - (eventsSinceFlush - writeIndex);
00141
00142         for (size_t i = 0; i < eventsSinceFlush; i++) {
00143             size_t idx = (startIdx + i) % EVENT_BUFFER_SIZE;
00144             fprintf(file, "%u;%lu;%u;%u\n",
00145                     events[idx].id,
00146                     events[idx].timestamp,
00147                     events[idx].group,
00148                     events[idx].event
00149             );
00150         }
00151         fclose(file);
00152         uart_print("Events saved to storage", VerboseLevel::INFO);
00153         return true;
00154     }
00155     return false;
00156 }
```

9.40 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` ,
 `PowerEvent::DISCHARGING = 0x09` , `PowerEvent::CHARGING = 0x0A` }

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

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

9.40.1 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](#).

9.40.2 Variable Documentation

9.40.2.1 timestamp

`uint32_t timestamp`

Timestamp of the event in milliseconds since boot.

Definition at line [2](#) of file [event_manager.h](#).

9.40.2.2 id

`uint16_t id`

Unique identifier for the event.

Definition at line [4](#) of file [event_manager.h](#).

9.40.2.3 group

`uint8_t group`

Event group.

Definition at line [6](#) of file [event_manager.h](#).

9.40.2.4 event

```
uint8_t event
```

Event code.

Definition at line 8 of file [event_manager.h](#).

9.41 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,
00105     DISCHARGING = 0x09,
00107     CHARGING = 0x0A,
00108 };
00109
00110
00116 enum class CommsEvent : uint8_t {
00118     RADIO_INIT = 0x01,
00120     RADIO_ERROR = 0x02,
00122     MSG_RECEIVED = 0x03,
00124     MSG_SENT = 0x04,
00126     UART_ERROR = 0x06
00127 };
00128
00134 enum class GPSEvent : uint8_t {
00136     LOCK = 0x01,
00138     LOST = 0x02,
00140     ERROR = 0x03,
00142     POWER_ON = 0x04,
00144     POWER_OFF = 0x05,
00146     DATA_READY = 0x06,
```

```

00148     PASS_THROUGH_START = 0x07,
00150     PASS_THROUGH_END = 0x08
00151 };
00152
00153 enum class ClockEvent : uint8_t {
00154     CHANGED = 0x01,
00155     GPS_SYNC = 0x02,
00156     GPS_SYNC_DATA_NOT_READY = 0x03
00157 };
00158
00159
00160
00161
00162
00163
00164
00165
00166
00167
00168
00169
00170
00171
00172
00173 class EventLog {
00174 public:
00175     uint32_t timestamp;
00176     uint16_t id;
00177     uint8_t group;
00178     uint8_t event;
00179 } __attribute__((packed));
00180
00181
00182
00183
00184
00185
00186
00187
00188
00189
00190
00191
00192
00193
00194 class EventManager {
00195 private:
00196     EventLog events[EVENT_BUFFER_SIZE];
00197     size_t eventCount;
00198     size_t writeIndex;
00199     mutex_t eventMutex;
00200     uint16_t nextEventId;
00201     size_t eventsSinceFlush;
00202
00203     EventManager() :
00204         eventCount(0),
00205         writeIndex(0),
00206         nextEventId(0),
00207         eventsSinceFlush(0)
00208     {
00209         mutex_init(&eventMutex);
00210     }
00211
00212     EventManager(const EventManager&) = delete;
00213     EventManager& operator=(const EventManager&) = delete;
00214
00215 public:
00216     static EventManager& get_instance() {
00217         static EventManager instance;
00218         return instance;
00219     }
00220
00221     bool init();
00222
00223     void log_event(uint8_t group, uint8_t event);
00224
00225     const EventLog& get_event(size_t index) const;
00226
00227     size_t get_event_count() const { return eventCount; }
00228
00229     bool save_to_storage();
00230
00231 };
00232
00233
00234
00235
00236
00237
00238
00239
00240
00241
00242
00243
00244
00245
00246
00247
00248
00249
00250
00251
00252
00253
00254
00255
00256 };
00257
00258
00259
00260
00261
00262
00263
00264 class EventEmitter {
00265 public:
00266     template<typename T>
00267     static void emit(EventGroup group, T event) {
00268         EventManager::get_instance().log_event(
00269             static_cast<uint8_t>(group),
00270             static_cast<uint8_t>(event)
00271         );
00272     }
00273 };
00274
00275
00276
00277
00278
00279
00280
00281 #endif // EVENT_MANAGER_H

```

9.42 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.

9.42.1 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](#).

9.43 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_gps_collection_paused()) {
00062         return;
00063     }
00064
00065     std::array<char, MAX_RAW_DATA_LENGTH> raw_data_buffer;
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.data());
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             if (raw_data_index < MAX_RAW_DATA_LENGTH - 1) {
00090                 raw_data_buffer[raw_data_index++] = c;
00091             } else {
00092                 raw_data_index = 0;
00093             }
00094         }
00095     }
00096 }

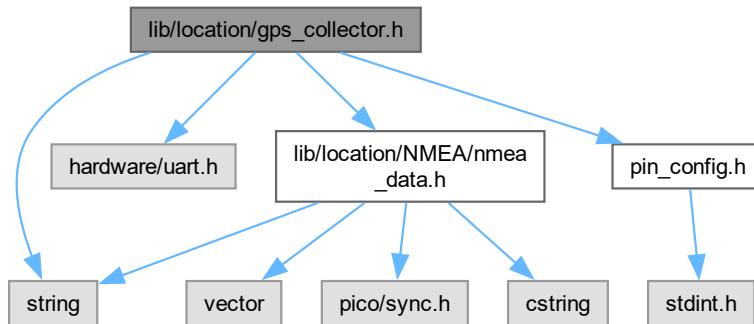
```

9.44 lib/location/gps_collector.h File Reference

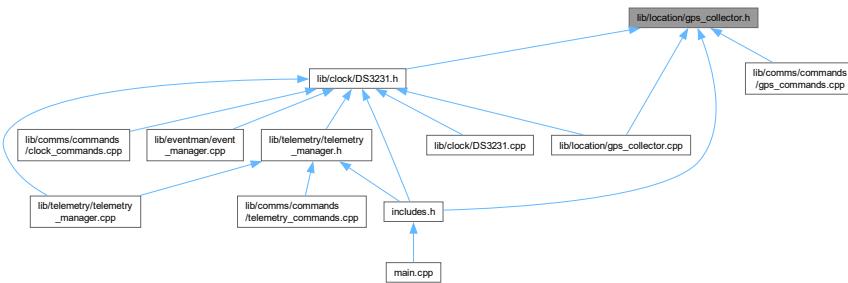
Header file for the GPS data collector module.

```
#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.

9.44.1 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](#).

9.45 gps_collector.h

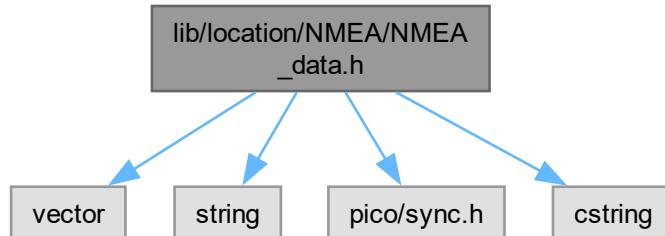
[Go to the documentation of this file.](#)

```
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
```

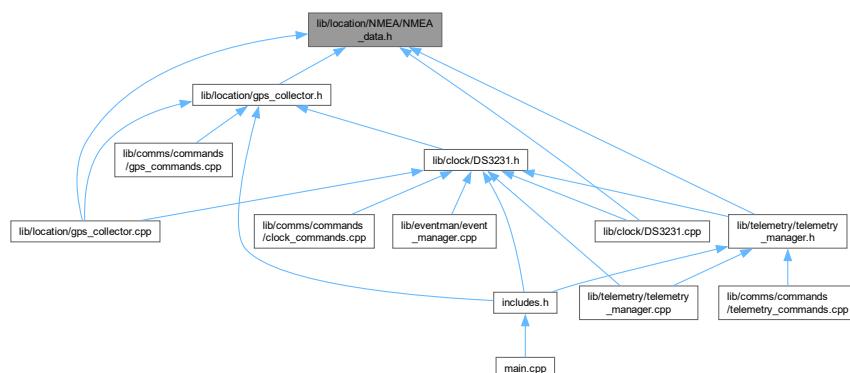
9.46 lib/location/NMEA/NMEA_data.h File Reference

Header file for the [NMEAData](#) class, which manages parsed NMEA sentences.

```
#include <vector>
#include <string>
#include "pico/sync.h"
#include <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.

9.46.1 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](#).

9.47 NMEA_data.h

[Go to the documentation of this file.](#)

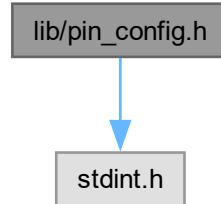
```

00001
00015
00016 #ifndef NMEA_DATA_H
00017 #define NMEA_DATA_H
00018
00019 #include <vector>
00020 #include <string>
00021 #include "pico/sync.h"
00022 #include <cstring>
00023
00032 class NMEAData {
00033 private:
00035     std::vector<std::string> rmc_tokens_;
00037     std::vector<std::string> gga_tokens_;
00039     mutex_t rmc_mutex_;
00041     mutex_t gga_mutex_;
00042
00047     NMEAData() {
00048         mutex_init(&rmc_mutex_);
00049         mutex_init(&gga_mutex_);
00050     }
00051
00055     NMEAData(const NMEAData&) = delete;
00059     NMEAData& operator=(const NMEAData&) = delete;
00060
00061 public:
00066     static NMEAData& get_instance() {
00067         static NMEAData instance;
00068         return instance;
00069     }
00070
00075     void update_rmc_tokens(const std::vector<std::string>& tokens) {
00076         mutex_enter_blocking(&rmc_mutex_);
00077         rmc_tokens_ = tokens;
00078         mutex_exit(&rmc_mutex_);
00079     }
00080
00085     void update_gga_tokens(const std::vector<std::string>& tokens) {
00086         mutex_enter_blocking(&gga_mutex_);
00087         gga_tokens_ = tokens;
00088         mutex_exit(&gga_mutex_);
00089     }
00090
00095     std::vector<std::string> get_rmc_tokens() const {
00096         mutex_enter_blocking(const_cast<mutex_t*>(&rmc_mutex_));
00097         std::vector<std::string> copy = rmc_tokens_;
00098         mutex_exit(const_cast<mutex_t*>(&rmc_mutex_));
00099         return copy;
00100     }
00101
00106     std::vector<std::string> get_gga_tokens() const {
00107         mutex_enter_blocking(const_cast<mutex_t*>(&gga_mutex_));
00108         std::vector<std::string> copy = gga_tokens_;
00109         mutex_exit(const_cast<mutex_t*>(&gga_mutex_));
00110         return copy;
00111     }
00112 };
00113
00114 #endif

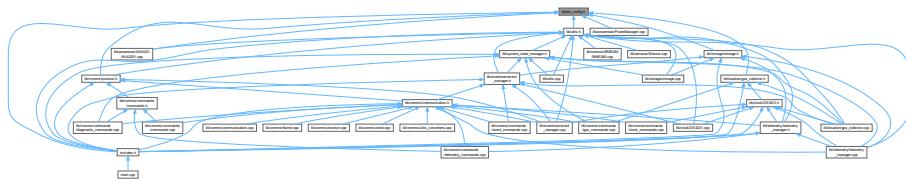
```

9.48 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 SENSORS_POWER_ENABLE_PIN 15
- #define SENSORS_I2C_PORT i2c0
- #define SENSORS_I2C_SDA_PIN 4
- #define SENSORS_I2C_SCL_PIN 5
- #define BUFFER_SIZE 85
- #define SD_SPI_PORT spi1
- #define SD_MISO_PIN 12
- #define SD_MOSI_PIN 11
- #define SD_SCK_PIN 10

- #define SD_CS_PIN 13
- #define SD_CARD_DETECT_PIN 28
- #define SX1278_MISO 16
- #define SX1278_CS 17
- #define SX1278_SCK 18
- #define SX1278莫斯I 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
- constexpr int lora_address_local = 37
- constexpr int lora_address_remote = 21

9.48.1 Macro Definition Documentation

9.48.1.1 DEBUG_UART_PORT

```
#define DEBUG_UART_PORT uart0
```

Definition at line 8 of file [pin_config.h](#).

9.48.1.2 DEBUG_UART_BAUD_RATE

```
#define DEBUG_UART_BAUD_RATE 115200
```

Definition at line 9 of file [pin_config.h](#).

9.48.1.3 DEBUG_UART_TX_PIN

```
#define DEBUG_UART_TX_PIN 0
```

Definition at line 11 of file [pin_config.h](#).

9.48.1.4 DEBUG_UART_RX_PIN

```
#define DEBUG_UART_RX_PIN 1
```

Definition at line 12 of file [pin_config.h](#).

9.48.1.5 MAIN_I2C_PORT

```
#define MAIN_I2C_PORT i2c1
```

Definition at line 14 of file [pin_config.h](#).

9.48.1.6 MAIN_I2C_SDA_PIN

```
#define MAIN_I2C_SDA_PIN 6
```

Definition at line 15 of file [pin_config.h](#).

9.48.1.7 MAIN_I2C_SCL_PIN

```
#define MAIN_I2C_SCL_PIN 7
```

Definition at line 16 of file [pin_config.h](#).

9.48.1.8 GPS_UART_PORT

```
#define GPS_UART_PORT uart1
```

Definition at line 19 of file [pin_config.h](#).

9.48.1.9 GPS_UART_BAUD_RATE

```
#define GPS_UART_BAUD_RATE 9600
```

Definition at line 20 of file [pin_config.h](#).

9.48.1.10 GPS_UART_TX_PIN

```
#define GPS_UART_TX_PIN 8
```

Definition at line 21 of file [pin_config.h](#).

9.48.1.11 GPS_UART_RX_PIN

```
#define GPS_UART_RX_PIN 9
```

Definition at line 22 of file [pin_config.h](#).

9.48.1.12 GPS_POWER_ENABLE_PIN

```
#define GPS_POWER_ENABLE_PIN 14
```

Definition at line 23 of file [pin_config.h](#).

9.48.1.13 SENSORS_POWER_ENABLE_PIN

```
#define SENSORS_POWER_ENABLE_PIN 15
```

Definition at line [25](#) of file [pin_config.h](#).

9.48.1.14 SENSORS_I2C_PORT

```
#define SENSORS_I2C_PORT i2c0
```

Definition at line [26](#) of file [pin_config.h](#).

9.48.1.15 SENSORS_I2C_SDA_PIN

```
#define SENSORS_I2C_SDA_PIN 4
```

Definition at line [27](#) of file [pin_config.h](#).

9.48.1.16 SENSORS_I2C_SCL_PIN

```
#define SENSORS_I2C_SCL_PIN 5
```

Definition at line [28](#) of file [pin_config.h](#).

9.48.1.17 BUFFER_SIZE

```
#define BUFFER_SIZE 85
```

Definition at line [30](#) of file [pin_config.h](#).

9.48.1.18 SD_SPI_PORT

```
#define SD_SPI_PORT spi1
```

Definition at line [33](#) of file [pin_config.h](#).

9.48.1.19 SD_MISO_PIN

```
#define SD_MISO_PIN 12
```

Definition at line [34](#) of file [pin_config.h](#).

9.48.1.20 SD_MOSI_PIN

```
#define SD_MOSI_PIN 11
```

Definition at line [35](#) of file [pin_config.h](#).

9.48.1.21 SD_SCK_PIN

```
#define SD_SCK_PIN 10
```

Definition at line [36](#) of file [pin_config.h](#).

9.48.1.22 SD_CS_PIN

```
#define SD_CS_PIN 13
```

Definition at line [37](#) of file [pin_config.h](#).

9.48.1.23 SD_CARD_DETECT_PIN

```
#define SD_CARD_DETECT_PIN 28
```

Definition at line [38](#) of file [pin_config.h](#).

9.48.1.24 SX1278_MISO

```
#define SX1278_MISO 16
```

Definition at line [40](#) of file [pin_config.h](#).

9.48.1.25 SX1278_CS

```
#define SX1278_CS 17
```

Definition at line [41](#) of file [pin_config.h](#).

9.48.1.26 SX1278_SCK

```
#define SX1278_SCK 18
```

Definition at line [42](#) of file [pin_config.h](#).

9.48.1.27 SX1278_MOSI

```
#define SX1278_MOSI 19
```

Definition at line [43](#) of file [pin_config.h](#).

9.48.1.28 SPI_PORT

```
#define SPI_PORT spi0
```

Definition at line [45](#) of file [pin_config.h](#).

9.48.1.29 READ_BIT

```
#define READ_BIT 0x80
```

Definition at line 46 of file [pin_config.h](#).

9.48.1.30 LORA_DEFAULT_SPI

```
#define LORA_DEFAULT_SPI spio
```

Definition at line 48 of file [pin_config.h](#).

9.48.1.31 LORA_DEFAULT_SPI_FREQUENCY

```
#define LORA_DEFAULT_SPI_FREQUENCY 8E6
```

Definition at line 49 of file [pin_config.h](#).

9.48.1.32 LORA_DEFAULT_SS_PIN

```
#define LORA_DEFAULT_SS_PIN 17
```

Definition at line 50 of file [pin_config.h](#).

9.48.1.33 LORA_DEFAULT_RESET_PIN

```
#define LORA_DEFAULT_RESET_PIN 22
```

Definition at line 51 of file [pin_config.h](#).

9.48.1.34 LORA_DEFAULT_DIO0_PIN

```
#define LORA_DEFAULT_DIO0_PIN 20
```

Definition at line 52 of file [pin_config.h](#).

9.48.1.35 PA_OUTPUT_RFO_PIN

```
#define PA_OUTPUT_RFO_PIN 11
```

Definition at line 54 of file [pin_config.h](#).

9.48.1.36 PA_OUTPUT_PA_BOOST_PIN

```
#define PA_OUTPUT_PA_BOOST_PIN 12
```

Definition at line 55 of file [pin_config.h](#).

9.48.2 Variable Documentation

9.48.2.1 lora_cs_pin

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

Definition at line 57 of file [pin_config.h](#).

9.48.2.2 lora_reset_pin

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

Definition at line 58 of file [pin_config.h](#).

9.48.2.3 lora_irq_pin

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

Definition at line 59 of file [pin_config.h](#).

9.48.2.4 lora_address_local

```
int lora_address_local = 37 [inline], [constexpr]
```

Definition at line 61 of file [pin_config.h](#).

9.48.2.5 lora_address_remote

```
int lora_address_remote = 21 [inline], [constexpr]
```

Definition at line 62 of file [pin_config.h](#).

9.49 pin_config.h

[Go to the documentation of this file.](#)

```

00001
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 SENSORS_POWER_ENABLE_PIN 15
00026 #define SENSORS_I2C_PORT i2c0
00027 #define SENSORS_I2C_SDA_PIN 4
00028 #define SENSORS_I2C_SCL_PIN 5
00029
00030 #define BUFFER_SIZE 85
00031
00032 // SPI configuration for SD card
00033 #define SD_SPI_PORT spi1
00034 #define SD_MISO_PIN 12
00035 #define SD_MOSI_PIN 11
00036 #define SD_SCK_PIN 10
00037 #define SD_CS_PIN 13
00038 #define SD_CARD_DETECT_PIN 28
00039
00040 #define SX1278_MISO 16
00041 #define SX1278_CS 17
00042 #define SX1278_SCK 18
00043 #define SX1278_MOSI 19
00044
00045 #define SPI_PORT spi0
00046 #define READ_BIT 0x80
00047
00048 #define LORA_DEFAULT_SPI          spi0
00049 #define LORA_DEFAULT_SPI_FREQUENCY 8E6
00050 #define LORA_DEFAULT_SS_PIN        17
00051 #define LORA_DEFAULT_RESET_PIN     22
00052 #define LORA_DEFAULT_DIO0_PIN      20
00053
00054 #define PA_OUTPUT_RFO_PIN         11
00055 #define PA_OUTPUT_PA_BOOST_PIN    12
00056
00057 inline constexpr int lora_cs_pin = 17;           // LoRa radio chip select
00058 inline constexpr int lora_reset_pin = 22;         // LoRa radio reset
00059 inline constexpr int lora_irq_pin = 28;           // LoRa hardware interrupt pin
00060
00061 inline constexpr int lora_address_local = 37;      // address of this device
00062 inline constexpr int lora_address_remote = 21;       // destination to send to
00063
00064 #endif

```

9.50 lib/powerman/INA3221/INA3221.cpp File Reference

Implementation of the [INA3221](#) power monitor driver.

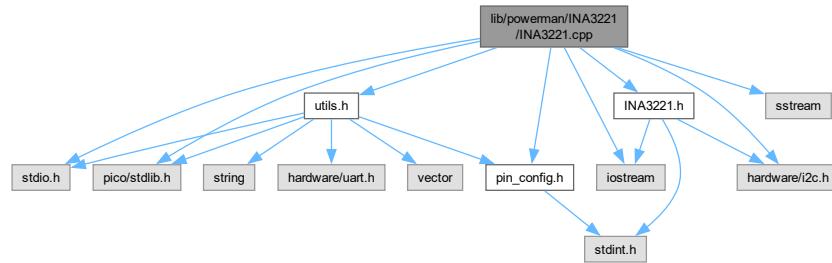
```

#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:



9.50.1 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](#).

9.51 INA3221.cpp

[Go to the documentation of this file.](#)

```

00001 #include "INA3221.h"
00002 #include <stdio.h>
00003 #include "pico/stdcib.h"
00004 #include "hardware/i2c.h"
00005 #include <iostream>
00006 #include "pin_config.h"
00007 #include "utils.h"
00008 #include <sstream>
00009
00010
00017
00018
00033
00034
00041 INA3221::INA3221(in3221_addr_t addr, i2c_inst_t* i2c)
00042 : _i2c_addr(addr), _i2c(i2c) {}
00043
00044
00051 bool INA3221::begin() {
00052     uart_print("INA3221 initializing...", VerboseLevel::DEBUG);
00053
00054     _shuntRes[0] = 10;
00055     _shuntRes[1] = 10;
00056     _shuntRes[2] = 10;
00057
00058     _filterRes[0] = 10;
00059     _filterRes[1] = 10;
00060     _filterRes[2] = 10;
00061
00062     uint16_t manuf_id = get_manufacturer_id();
00063     uint16_t die_id = get_die_id();
00064     std::stringstream ss;
00065     ss << "INA3221 Manufacturer ID: 0x" << std::hex << manuf_id
  
```

```

00066             << ", Die ID: 0x" << die_id;
00067             uart_print(ss.str(), VerbosityLevel::INFO);
00068
00069     if (manuf_id == 0x5449 && die_id == 0x3220) {
00070         uart_print("INA3221 found and initialized.", VerbosityLevel::DEBUG);
00071         return true;
00072     } else {
00073         uart_print("INA3221 initialization failed. Incorrect IDs.", VerbosityLevel::ERROR);
00074         return false;
00075     }
00076 }
00077
00078 uint16_t INA3221::get_manufacturer_id() {
00079     uint16_t id = 0;
00080     _read(INA3221_REG_MANUF_ID, &id);
00081     return id;
00082 }
00083
00084 uint16_t INA3221::get_die_id() {
00085     uint16_t id = 0;
00086     _read(INA3221_REG_DIE_ID, &id);
00087     return id;
00088 }
00089
00090 uint16_t INA3221::read_register(ina3221_reg_t reg) {
00091     uint16_t val = 0;
00092     _read(reg, &val);
00093     return val;
00094 }
00095
00096 //configure
00097
00098 void INA3221::set_mode_continuous() {
00099     conf_reg_t conf_reg;
00100
00101     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00102     conf_reg.mode_continious_en = 1;
00103     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00104 }
00105
00106
00107 void INA3221::set_mode_triggered() {
00108     conf_reg_t conf_reg;
00109
00110     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00111     conf_reg.mode_continious_en = 0;
00112     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00113 }
00114
00115 void INA3221::set_averaging_mode(ina3221_avg_mode_t mode) {
00116     conf_reg_t conf_reg;
00117
00118     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00119     conf_reg.avg_mode = mode;
00120     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00121 }
00122
00123 //get measurement
00124 int32_t INA3221::get_shunt_voltage(ina3221_ch_t channel) {
00125     int32_t res;
00126     ina3221_reg_t reg;
00127     uint16_t val_raw = 0;
00128
00129     switch(channel){
00130         case INA3221_CH1:
00131             reg = INA3221_REG_CH1_SHUNTV;
00132             break;
00133         case INA3221_CH2:
00134             reg = INA3221_REG_CH2_SHUNTV;
00135             break;
00136         case INA3221_CH3:
00137             reg = INA3221_REG_CH3_SHUNTV;
00138             break;
00139     }
00140
00141     _read(reg, &val_raw);
00142
00143     res = (int16_t) (val_raw >> 3);
00144     res *= SHUNT_VOLTAGE_LSB_UV;
00145
00146     return res;
00147 }
00148
00149

```

```

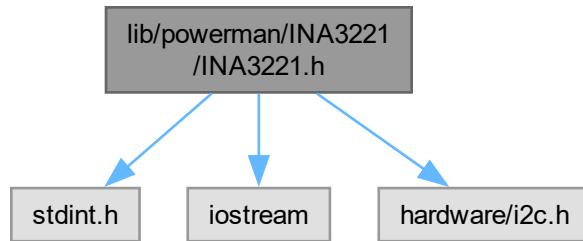
00196 float INA3221::get_current_ma(ina3221_ch_t channel) {
00197     int32_t shunt_uV = 0;
00198     float current_A = 0;
00199
00200     shunt_uV = get_shunt_voltage(channel);
00201     current_A = shunt_uV / (int32_t)_shuntRes[channel];
00202     return current_A;
00203 }
00204
00205
00212 float INA3221::get_voltage(ina3221_ch_t channel) {
00213     float voltage_V = 0.0;
00214     ina3221_reg_t reg;
00215     uint16_t val_raw = 0;
00216
00217     switch(channel){
00218         case INA3221_CH1:
00219             reg = INA3221_REG_CH1_BUSV;
00220             break;
00221         case INA3221_CH2:
00222             reg = INA3221_REG_CH2_BUSV;
00223             break;
00224         case INA3221_CH3:
00225             reg = INA3221_REG_CH3_BUSV;
00226             break;
00227     }
00228
00229     _read(reg, &val_raw);
00230     voltage_V = val_raw / 1000.0;
00231     return voltage_V;
00232 }
00233
00234
00235 // private
00242 void INA3221::_read(ina3221_reg_t reg, uint16_t *val) {
00243     uint8_t reg_buf = reg;
00244     uint8_t data[2];
00245
00246     int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, &reg_buf, 1, true);
00247     if (ret != 1) {
00248         std::cerr << "Failed to write register address to I2C device." << std::endl;
00249         return;
00250     }
00251
00252     ret = i2c_read_blocking(MAIN_I2C_PORT, _i2c_addr, data, 2, false);
00253     if (ret != 2) {
00254         std::cerr << "Failed to read data from I2C device." << std::endl;
00255         return;
00256     }
00257
00258     *val = (data[0] << 8) | data[1];
00259 }
00260
00261
00268 void INA3221::_write(ina3221_reg_t reg, uint16_t *val) {
00269     uint8_t buf[3];
00270     buf[0] = reg;
00271     buf[1] = (*val >> 8) & 0xFF; // MSB
00272     buf[2] = (*val) & 0xFF; // LSB
00273
00274     int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, buf, 3, false);
00275     if (ret != 3) {
00276         std::cerr << "Failed to write data to I2C device." << std::endl;
00277     }
00278 }
```

9.52 lib/powerman/INA3221/INA3221.h File Reference

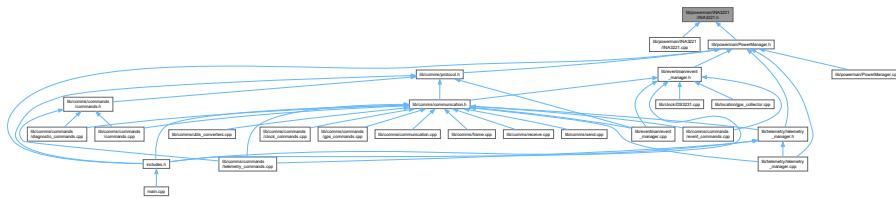
Header file for the [INA3221](#) triple-channel power monitor driver.

```
#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_ALEF` ,
`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_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_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.

 - enum [ina3221_op_mode_t](#) { [CONTINUOUS_MODE](#) , [TRIGGERED_MODE](#) }

Operating 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.

9.52.1 Detailed Description

Header file for the [INA3221](#) triple-channel power monitor driver.

Definition in file [INA3221.h](#).

9.52.2 Enumeration Type Documentation

9.52.2.1 [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](#).

9.52.2.2 [ina3221_ch_t](#)

enum [ina3221_ch_t](#)

Enumerator

INA3221_CH1	
INA3221_CH2	
INA3221_CH3	

Definition at line 19 of file [INA3221.h](#).

9.52.2.3 [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 33 of file [INA3221.h](#).

9.52.2.4 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 61 of file [INA3221.h](#).

9.52.2.5 ina3221_op_mode_t

```
enum ina3221_op_mode_t
```

Operating mode settings.

Continuous or triggered (single-shot) mode

Enumerator

CONTINUOUS_MODE	
TRIGGERED_MODE	

Definition at line 76 of file [INA3221.h](#).

9.52.3 Variable Documentation

9.52.3.1 INA3221_CH_NUM

```
const int INA3221_CH_NUM = 3
```

Number of channels in [INA3221](#).

Definition at line 26 of file [INA3221.h](#).

9.52.3.2 SHUNT_VOLTAGE_LSB_UV

```
const int SHUNT_VOLTAGE_LSB_UV = 5
```

LSB value for shunt voltage measurements in microvolts.

Definition at line 28 of file [INA3221.h](#).

9.53 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
00019 typedef enum {
00020     INA3221_CH1 = 0,
00021     INA3221_CH2,
00022     INA3221_CH3,
00023 } ina3221_ch_t;
00024
00026 const int INA3221_CH_NUM = 3;
00028 const int SHUNT_VOLTAGE_LSB_UV = 5;
00029
00033 typedef enum {
00034     INA3221_REG_CONF = 0,
00035     INA3221_REG_CH1_SHUNTV,
00036     INA3221_REG_CH1_BUSV,
00037     INA3221_REG_CH2_SHUNTV,
00038     INA3221_REG_CH2_BUSV,
00039     INA3221_REG_CH3_SHUNTV,
00040     INA3221_REG_CH3_BUSV,
00041     INA3221_REG_CH1_CRIT_ALERT_LIM,
00042     INA3221_REG_CH1_WARNING_ALERT_LIM,
```

```

00043     INA3221_REG_CH2_CRIT_ALERT_LIM,
00044     INA3221_REG_CH2_WARNING_ALERT_LIM,
00045     INA3221_REG_CH3_CRIT_ALERT_LIM,
00046     INA3221_REG_CH3_WARNING_ALERT_LIM,
00047     INA3221_REG_SHUNTV_SUM,
00048     INA3221_REG_SHUNTV_SUM_LIM,
00049     INA3221_REG_MASK_ENABLE,
00050     INA3221_REG_PWR_VALID_HI_LIM,
00051     INA3221_REG_PWR_VALID_LO_LIM,
00052     INA3221_REG_MANUF_ID = 0xFE,
00053     INA3221_REG_DIE_ID = 0xFF
00054 } ina3221_reg_t;
00055
00056
00061 typedef enum {
00062     INA3221_REG_CONF_AVG_1 = 0,
00063     INA3221_REG_CONF_AVG_4,
00064     INA3221_REG_CONF_AVG_16,
00065     INA3221_REG_CONF_AVG_64,
00066     INA3221_REG_CONF_AVG_128,
00067     INA3221_REG_CONF_AVG_256,
00068     INA3221_REG_CONF_AVG_512,
00069     INA3221_REG_CONF_AVG_1024
00070 } ina3221_avg_mode_t;
00071
00076 typedef enum {
00077     CONTINUOUS_MODE,
00078     TRIGGERED_MODE
00079 } ina3221_op_mode_t;
00080
00086 class INA3221 {
00087
00091     typedef struct {
00092         uint16_t mode_shunt_en:1;
00093         uint16_t mode_bus_en:1;
00094         uint16_t mode_continious_en:1;
00095         uint16_t shunt_conv_time:3;
00096         uint16_t bus_conv_time:3;
00097         uint16_t avg_mode:3;
00098         uint16_t ch3_en:1;
00099         uint16_t ch2_en:1;
00100         uint16_t ch1_en:1;
00101         uint16_t reset:1;
00102     } conf_reg_t;
00103
00107     typedef struct {
00108         uint16_t conv_ready:1;
00109         uint16_t timing_ctrl_alert:1;
00110         uint16_t pwr_valid_alert:1;
00111         uint16_t warn_alert_ch3:1;
00112         uint16_t warn_alert_ch2:1;
00113         uint16_t warn_alert_ch1:1;
00114         uint16_t shunt_sum_alert:1;
00115         uint16_t crit_alert_ch3:1;
00116         uint16_t crit_alert_ch2:1;
00117         uint16_t crit_alert_ch1:1;
00118         uint16_t crit_alert_latch_en:1;
00119         uint16_t warn_alert_latch_en:1;
00120         uint16_t shunt_sum_en_ch3:1;
00121         uint16_t shunt_sum_en_ch2:1;
00122         uint16_t shunt_sum_en_ch1:1;
00123         uint16_t reserved:1;
00124     } masken_reg_t;
00125
00126 // I2C address
00127 ina3221_addr_t _i2c_addr;
00128 i2c_inst_t* _i2c;
00129
00130 // Shunt resistance in mOhm
00131 uint32_t _shuntRes[INA3221_CH_NUM];
00132
00133 // Series filter resistance in Ohm
00134 uint32_t _filterRes[INA3221_CH_NUM];
00135
00136 // Value of Mask/Enable register.
00137 masken_reg_t _masken_reg;
00138
00139 // Reads 16 bytes from a register.
00140 void _read(ina3221_reg_t reg, uint16_t *val);
00141
00142 // Writes 16 bytes to a register.
00143 void _write(ina3221_reg_t reg, uint16_t *val);
00144
00145 public:
00146     INA3221(ina3221_addr_t addr, i2c_inst_t* i2c);
00147     // Initializes INA3221

```

```

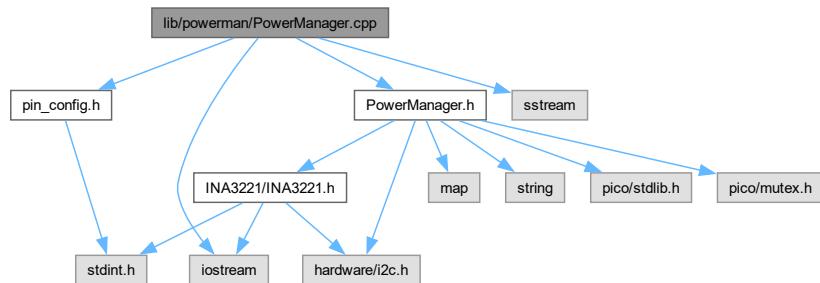
00149     bool begin();
00150
00151     // Gets a register value.
00152     uint16_t read_register(ina3221_reg_t reg);
00153
00154     // Sets operating mode to continuous
00155     void set_mode_continuous();
00156
00157     // Sets operating mode to triggered (single-shot)
00158     void set_mode_triggered();
00159
00160     // Sets averaging mode. Sets number of samples that are collected
00161     // and averaged together.
00162     void set_averaging_mode(ina3221_avg_mode_t mode);
00163
00164     // Gets manufacturer ID.
00165     // Should read 0x5449.
00166     uint16_t get_manufacturer_id();
00167
00168     // Gets die ID.
00169     // Should read 0x3220.
00170     uint16_t get_die_id();
00171
00172     // Gets shunt voltage in uV.
00173     int32_t get_shunt_voltage(ina3221_ch_t channel);
00174
00175     // Gets current in mA.
00176     float get_current_ma(ina3221_ch_t channel);
00177
00178     // Gets bus voltage in V.
00179     float get_voltage(ina3221_ch_t channel);
00180 };
00181
00182 #endif

```

9.54 lib/powerman/PowerManager.cpp File Reference

Implementation of the [PowerManager](#) class, which manages power-related functions.

```
#include "PowerManager.h"
#include <iostream>
#include <sstream>
#include "pin_config.h"
Include dependency graph for PowerManager.cpp:
```



9.54.1 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](#).

9.55 PowerManager.cpp

[Go to the documentation of this file.](#)

```

0001
00014
00015 #include "PowerManager.h"
00016 #include <iostream>
00017 #include <sstream>
00018 #include "pin_config.h"
00019
00025 PowerManager::PowerManager()
00026     : ina3221_(INA3221_ADDR40_GND, MAIN_I2C_PORT) {
00027     recursive_mutex_init(&powerman_mutex_);
00028 }
00029
00035 PowerManager& PowerManager::get_instance() {
00036     static PowerManager instance;
00037     return instance;
00038 }
00039
00045 bool PowerManager::initialize() {
00046     recursive_mutex_enter_blocking(&powerman_mutex_);
00047     initialized_ = ina3221_.begin();
00048
00049     recursive_mutex_exit(&powerman_mutex_);
00050     return initialized_;
00051 }
00052
00058 std::string PowerManager::read_device_ids() {
00059     if (!initialized_) return "noinit";
00060     recursive_mutex_enter_blocking(&powerman_mutex_);
00061     std::stringstream man_ss;
00062     man_ss << std::hex << ina3221_.get_manufacturer_id();
00063     std::string MAN = "MAN 0x" + man_ss.str();
00064
00065     std::stringstream die_ss;
00066     die_ss << std::hex << ina3221_.get_die_id();
00067     std::string DIE = "DIE 0x" + die_ss.str();
00068     recursive_mutex_exit(&powerman_mutex_);
00069     return MAN + " - " + DIE;
00070 }
00071
00077 float PowerManager::get_voltage_battery() {
00078     if (!initialized_) return 0.0f;
00079     recursive_mutex_enter_blocking(&powerman_mutex_);
00080     float voltage = ina3221_.get_voltage(INA3221_CH1);
00081     recursive_mutex_exit(&powerman_mutex_);
00082     return voltage;
00083 }
00084
00090 float PowerManager::get_voltage_5v() {
00091     if (!initialized_) return 0.0f;
00092     recursive_mutex_enter_blocking(&powerman_mutex_);
00093     float voltage = ina3221_.get_voltage(INA3221_CH2);
00094     recursive_mutex_exit(&powerman_mutex_);
00095     return voltage;
00096 }
00097
00103 float PowerManager::get_voltage_solar() {
00104     if (!initialized_) return 0.0f;
00105     recursive_mutex_enter_blocking(&powerman_mutex_);
00106     float voltage = ina3221_.get_voltage(INA3221_CH3);
00107     recursive_mutex_exit(&powerman_mutex_);
00108     return voltage;
00109 }
00110
00116 float PowerManager::get_current_charge_usb() {
00117     if (!initialized_) return 0.0f;
00118     recursive_mutex_enter_blocking(&powerman_mutex_);
00119     float current = ina3221_.get_current_ma(INA3221_CH1);
00120     recursive_mutex_exit(&powerman_mutex_);
00121     return current;
00122 }
00123
00129 float PowerManager::get_current_draw() {
00130     if (!initialized_) return 0.0f;
00131     recursive_mutex_enter_blocking(&powerman_mutex_);
00132     float current = ina3221_.get_current_ma(INA3221_CH2);
00133     recursive_mutex_exit(&powerman_mutex_);
00134     return current;
00135 }
00136
00142 float PowerManager::get_current_charge_solar() {
00143     if (!initialized_) return 0.0f;
00144     recursive_mutex_enter_blocking(&powerman_mutex_);

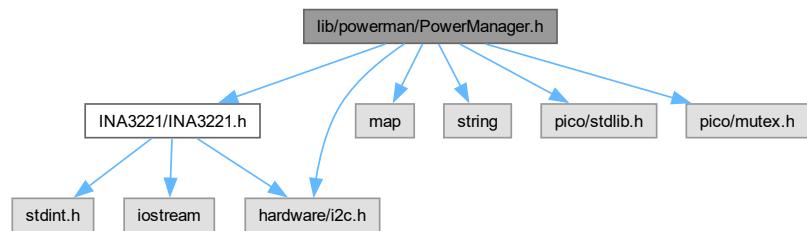
```

```
00145     float current = ina3221_.get_current_ma(INA3221_CH3);
00146     recursive_mutex_exit(&powerman_mutex_);
00147     return current;
00148 }
00149
00155 float PowerManager::get_current_charge_total() {
00156     if (!initialized_) return 0.0f;
00157     recursive_mutex_enter_blocking(&powerman_mutex_);
00158     float current = ina3221_.get_current_ma(INA3221_CH1) + ina3221_.get_current_ma(INA3221_CH3);
00159     recursive_mutex_exit(&powerman_mutex_);
00160     return current;
00161 }
00162
00169 void PowerManager::configure(ina3221_op_mode_t op_mode, ina3221_avg_mode_t avg_mode) {
00170     if (!initialized_) return;
00171
00172     recursive_mutex_enter_blocking(&powerman_mutex_);
00173
00174     if (op_mode == CONTINUOUS_MODE) {
00175         ina3221_.set_mode_continuous();
00176     } else {
00177         ina3221_.set_mode_triggered();
00178     }
00179
00180     ina3221_.set_averaging_mode(avg_mode);
00181
00182     recursive_mutex_exit(&powerman_mutex_);
00183 }
```

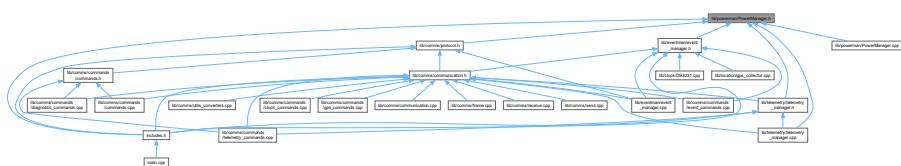
9.56 lib/powerman/PowerManager.h File Reference

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.

9.56.1 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](#).

9.57 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
00025 class PowerManager {
00026 public:
00027     PowerManager(i2c_inst_t* i2c);
00028
00029     static PowerManager& get_instance();
00030
00031     bool initialize();
00032
00033     std::string read_device_ids();
00034
00035     float get_current_charge_solar();
00036
00037     float get_current_charge_usb();
00038
00039     float get_current_charge_total();
00040
00041     float get_current_draw();
00042
00043     float get_voltage_battery();
00044
00045     float get_voltage_5v();
00046
00047     float get_voltage_solar();
00048
00049     void configure(ina3221_op_mode_t op_mode, ina3221_avg_mode_t avg_mode);
00050
00051
00052     static constexpr float SOLAR_CURRENT_THRESHOLD = 50.0f; // mA
00053     static constexpr float USB_CURRENT_THRESHOLD = 50.0f; // mA
00054     static constexpr float BATTERY_LOW_THRESHOLD = 2.8f; // V
00055     static constexpr float BATTERY_FULL_THRESHOLD = 4.2f; // V
00056
00057 private:
00058     INA3221 ina3221_;
00059     bool initialized_;
00060     recursive_mutex_t powerman_mutex_;
00061
00062     PowerManager();
00063
00064     PowerManager(const PowerManager&) = delete;
00065     PowerManager& operator=(const PowerManager&) = delete;
00066 };
00067
00068 #endif // POWER_MANAGER_H

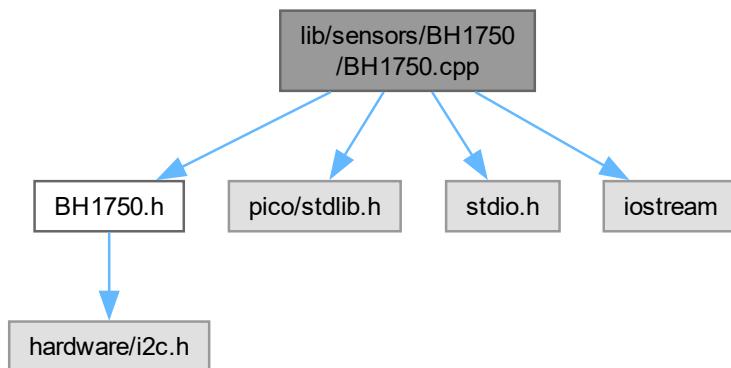
```

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

Implementation of the [BH1750](#) light sensor class.

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

Include dependency graph for BH1750.cpp:



9.58.1 Detailed Description

Implementation of the [BH1750](#) light sensor class.

This file contains the implementation of the [BH1750](#) class, which provides an interface to the [BH1750](#) digital light sensor using the I2C communication protocol.

Definition in file [BH1750.cpp](#).

9.59 BH1750.cpp

[Go to the documentation of this file.](#)

```

00001
00008
00009 #include "BH1750.h"
00010 #include "pico/stdlib.h"
00011 #include <stdio.h>
00012 #include <iostream>
00013
00020 BH1750::BH1750(i2c_inst_t* i2c, uint8_t addr) : _i2c_addr(addr), _i2c_port_(i2c) {}
00021
00028 bool BH1750::begin(Mode mode) {
00029     write8(static_cast<uint8_t>(Mode::POWER_ON));
00030     write8(static_cast<uint8_t>(Mode::RESET));
00031     bool config_status = configure(mode);
00032
00033     return config_status;
00034 }
00035
  
```

```

00042 bool BH1750::configure(Mode mode) {
00043     uint8_t modeVal = static_cast<uint8_t>(mode);
00044     switch (mode) {
00045         case Mode::UNCONFIGURED_POWER_DOWN:
00046         case Mode::POWER_ON:
00047         case Mode::RESET:
00048         case Mode::CONTINUOUS_HIGH_RES_MODE:
00049         case Mode::CONTINUOUS_HIGH_RES_MODE_2:
00050         case Mode::CONTINUOUS_LOW_RES_MODE:
00051         case Mode::ONE_TIME_HIGH_RES_MODE:
00052         case Mode::ONE_TIME_HIGH_RES_MODE_2:
00053         case Mode::ONE_TIME_LOW_RES_MODE:
00054             write8(modeVal);
00055             sleep_ms(10);
00056             return true;
00057     default:
00058         return false;
00059     }
00060 }
00061
00067 float BH1750::get_light_level() {
00068     uint8_t buffer[2];
00069     i2c_read_blocking(i2c_port_, _i2c_addr, buffer, 2, false);
00070     uint16_t level = (buffer[0] << 8) | buffer[1];
00071
00072     float lux = static_cast<float>(level) / 1.2f;
00073     return lux;
00074 }
00075
00081 void BH1750::write8(uint8_t data) {
00082     uint8_t buf[1] = {data};
00083     i2c_write_blocking(i2c_port_, _i2c_addr, buf, 1, false);
00084 }

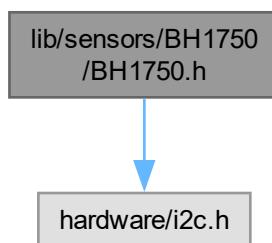
```

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

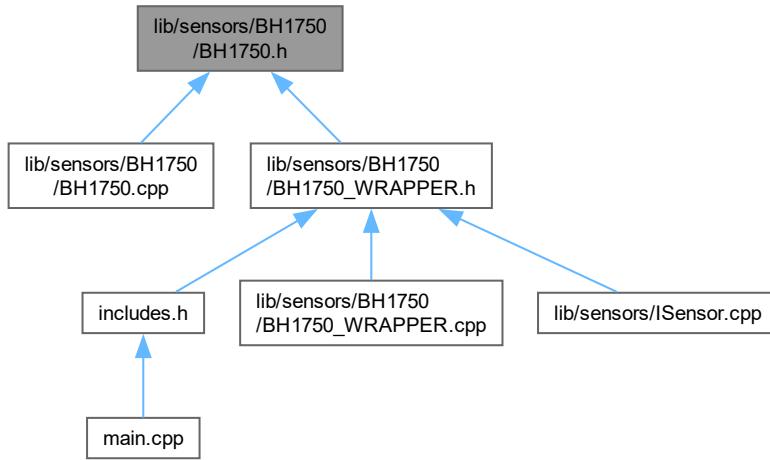
Header file for the [BH1750](#) light sensor class.

```
#include "hardware/i2c.h"
```

Include dependency graph for BH1750.h:



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



Classes

- class [BH1750](#)
Class to interface with the [BH1750](#) light sensor.

Macros

- `#define _BH1750_DEVICE_ID 0xE1`
Correct content of WHO_AM_I register (not actually used in this driver).
- `#define _BH1750_MTREG_MIN 31`
Minimum value for the MTREG register.
- `#define _BH1750_MTREG_MAX 254`
Maximum value for the MTREG register.
- `#define _BH1750_DEFAULT_MTREG 69`
Default value for the MTREG register.

9.60.1 Detailed Description

Header file for the [BH1750](#) light sensor class.

This class provides an interface to the [BH1750](#) digital light sensor using the I²C communication protocol.

Definition in file [BH1750.h](#).

9.61 BH1750.h

[Go to the documentation of this file.](#)

```

00001
00008
00009 #ifndef __BH1750_H__
00010 #define __BH1750_H__
00011
00012 #include "hardware/i2c.h"
00013
00019
00025
00031
00036 #define _BH1750_DEVICE_ID 0xE1
00037
00042 #define _BH1750_MTREG_MIN 31
00043
00048 #define _BH1750_MTREG_MAX 254
00049
00054 #define _BH1750_DEFAULT_MTREG 69
00055
00060 class BH1750 {
00061 public:
00066     enum class Mode : uint8_t {
00068         UNCONFIGURED_POWER_DOWN = 0x00,
00070         POWER_ON = 0x01,
00072         RESET = 0x07,
00074         CONTINUOUS_HIGH_RES_MODE = 0x10,
00076         CONTINUOUS_HIGH_RES_MODE_2 = 0x11,
00078         CONTINUOUS_LOW_RES_MODE = 0x13,
00080         ONE_TIME_HIGH_RES_MODE = 0x20,
00082         ONE_TIME_HIGH_RES_MODE_2 = 0x21,
00084         ONE_TIME_LOW_RES_MODE = 0x23
00085     };
00086
00092     BH1750(i2c_inst_t* i2c, uint8_t addr = 0x23);
00093
00099     bool begin(Mode mode = Mode::CONTINUOUS_HIGH_RES_MODE);
00100
00106     bool configure(Mode mode);
00107
00112     float get_light_level();
00113
00114 private:
00119     void write8(uint8_t data);
00120
00122     uint8_t _i2c_addr;
00124     i2c_inst_t* _i2c_port_;
00125 };
00126
00127 #endif // __BH1750_H__

```

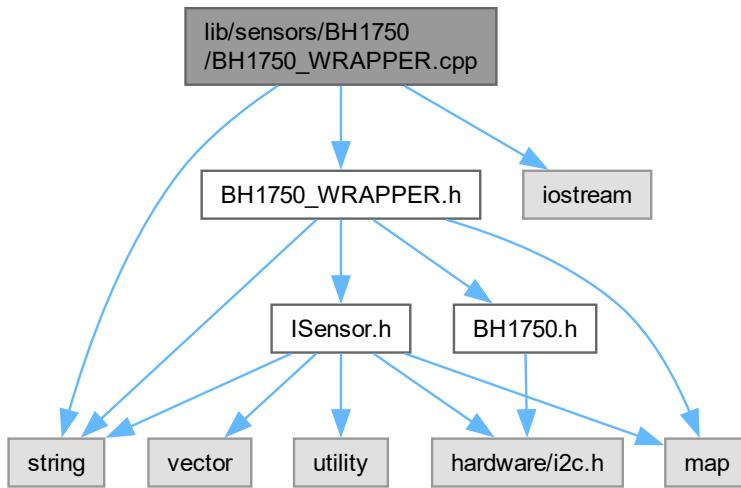
9.62 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference

```

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

```

Include dependency graph for BH1750_WRAPPER.cpp:



9.63 BH1750_WRAPPER.cpp

[Go to the documentation of this file.](#)

```

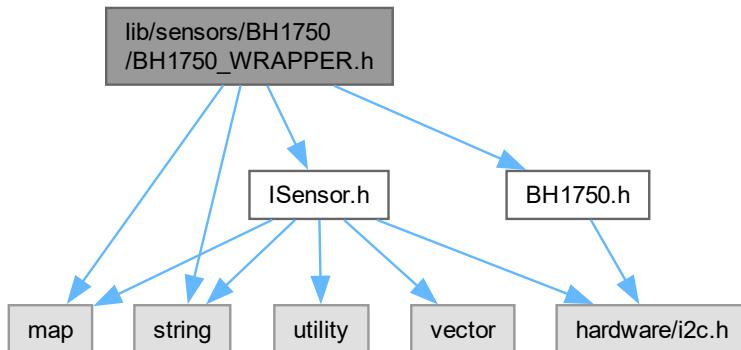
00001 #include "BH1750_WRAPPER.h"
00002 #include <string>
00003 #include <iostream>
00004
00005 BH1750Wrapper::BH1750Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {
00006     sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
00007 }
00008
00009 bool BH1750Wrapper::init() {
00010     initialized_ = sensor_.begin();
00011     return initialized_;
00012 }
00013
00014 float BH1750Wrapper::read_data(SensorDataTypeIdentifier type) {
00015     if (type == SensorDataTypeIdentifier::LIGHT_LEVEL) {
00016         return sensor_.get_light_level();
00017     }
00018     return 0.0f;
00019 }
00020
00021 bool BH1750Wrapper::is_initialized() const {
00022     return initialized_;
00023 }
00024
00025 SensorType BH1750Wrapper::get_type() const {
00026     return SensorType::LIGHT;
00027 }
00028
00029 bool BH1750Wrapper::configure(const std::map<std::string, std::string>& config) {
00030     for (const auto& [key, value] : config) {
00031         if (key == "measurement_mode") {
00032             if (value == "continuously_high_resolution") {
00033                 sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
00034             }
00035             else if (value == "continuously_high_resolution_2") {
00036                 sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE_2);
00037             }
00038             else if (value == "continuously_low_resolution") {
00039                 sensor_.configure(BH1750::Mode::CONTINUOUS_LOW_RES_MODE);
00040             }
00041             else if (value == "one_time_high_resolution") {
  
```

```

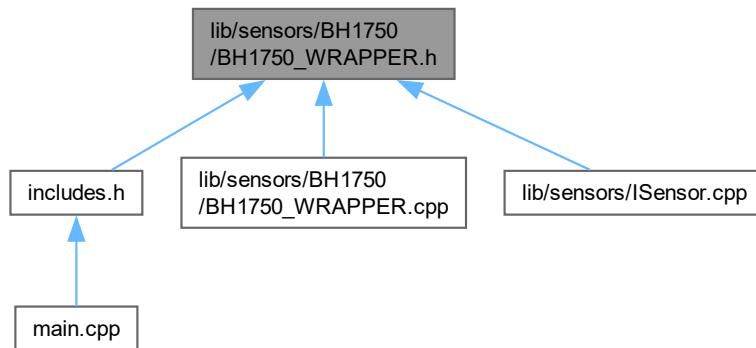
00042         sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE);
00043     }
00044     else if (value == "one_time_high_resolution_2") {
00045         sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE_2);
00046     }
00047     else if (value == "one_time_low_resolution") {
00048         sensor_.configure(BH1750::Mode::ONE_TIME_LOW_RES_MODE);
00049     }
00050     else {
00051         std::cerr << "[BH1750Wrapper] Unknown measurement_mode value: " << value << std::endl;
00052         return false;
00053     }
00054 }
00055 else {
00056     std::cerr << "[BH1750Wrapper] Unknown configuration key: " << key << std::endl;
00057     return false;
00058 }
00059 }
00060 return true;
00061 }
```

9.64 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](#)

9.65 BH1750_WRAPPER.h

[Go to the documentation of this file.](#)

```

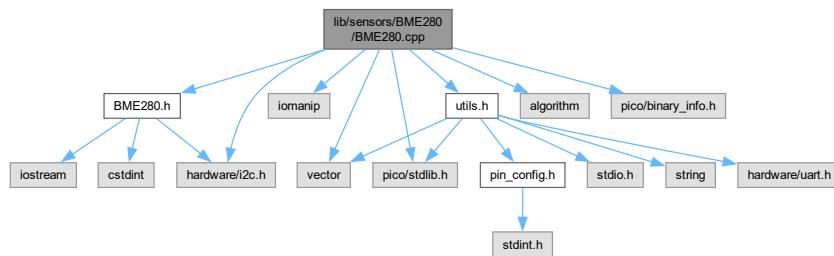
00001 #ifndef BH1750_WRAPPER_H
00002 #define BH1750_WRAPPER_H
00003
00004 #include "ISensor.h"
00005 #include "BH1750.h"
00006 #include <map>
00007 #include <string>
00008
00009 class BH1750Wrapper : public ISensor {
00010 private:
00011     BH1750 sensor_;
00012     bool initialized_ = false;
00013
00014 public:
00015     BH1750Wrapper(i2c_inst_t* i2c);
00016     BH1750Wrapper();
00017     int get_i2c_addr();
00018     bool init() override;
00019     float read_data(SensorDataTypeIdentifier type) override;
00020     bool is_initialized() const override;
00021     SensorType get_type() const override;
00022
00023     bool configure(const std::map<std::string, std::string>& config);
00024
00025     uint8_t get_address() const override {
00026         return 0x23;
00027     }
00028 };
00029 #endif // BH1750_WRAPPER_H
  
```

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

Implementation of the [BME280](#) environmental sensor class.

```
#include "BME280.h"
#include <iomanip>
#include <vector>
#include <algorithm>
#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdlib.h"
#include "utils.h"

Include dependency graph for BME280.cpp:
```



9.66.1 Detailed Description

Implementation of the [BME280](#) environmental sensor class.

This file contains the implementation of the [BME280](#) class, which provides an interface to the [BME280](#) temperature, pressure, and humidity sensor using the I2C communication protocol.

Definition in file [BME280.cpp](#).

9.67 BME280.cpp

[Go to the documentation of this file.](#)

```

00001
00009
00010 #include "BME280.h"
00011 #include <iomanip>
00012 #include <vector>
00013 #include <algorithm>
00014 #include "hardware/i2c.h"
00015 #include "pico/binary_info.h"
00016 #include "pico/stdlib.h"
00017 #include "utils.h"
00018
00024 BME280::BME280(i2c_inst_t* i2cPort, uint8_t address)
00025     : i2c_port(i2cPort), device_addr(address), calib_params{}, initialized_(false), t_fine(0) {
00026 }
00027
00032 bool BME280::init() {
00033     if (!i2c_port) {
00034         uart_print("BME280 I2C port not initialized.", VerbosityLevel::ERROR);
00035         return false;
00036     }
00037
00038     // Check device ID to confirm it's a BME280
00039     uint8_t chip_id;
00040     if (!read_register(0xD0, &chip_id)) {
00041         uart_print("Failed to read chip ID from BME280.", VerbosityLevel::ERROR);
00042         return false;
00043     }
00044 }
```

```

00045     if (chip_id != 0x60) {
00046         uart_print("Invalid BME280 chip ID.", VerbosityLevel::ERROR);
00047         return false;
00048     }
00049
00050     // Configure sensor
00051     if (!configure_sensor()) {
00052         uart_print("Failed to configure BME280 sensor.", VerbosityLevel::ERROR);
00053         return false;
00054     }
00055
00056     // Retrieve calibration parameters
00057     if (!get_calibration_parameters()) {
00058         uart_print("Failed to get calibration parameters from BME280.", VerbosityLevel::ERROR);
00059         return false;
00060     }
00061
00062     initialized_ = true;
00063     uart_print("BME280 initialized.", VerbosityLevel::INFO);
00064     return true;
00065 }
00066
00070 void BME280::reset() {
00071     write_register(REG_RESET, 0xB6);
00072     sleep_ms(10); // Wait for reset to complete
00073 }
00074
00082 bool BME280::read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity) {
00083     if (!initialized_) {
00084         uart_print("BME280 not initialized.", VerbosityLevel::ERROR);
00085         return false;
00086     }
00087
00088     // Define the starting register address
00089     uint8_t start_reg = REG_PRESSURE_MSB;
00090     // Total bytes to read: 3 (pressure) + 3 (temperature) + 2 (humidity) = 8
00091     uint8_t buf[8] = {0};
00092
00093     // Write the starting register address
00094     if (!write_register(start_reg, 1)) {
00095         uart_print("Failed to write to BME280.", VerbosityLevel::ERROR);
00096         return false;
00097     }
00098
00099     // Read data
00100    int ret = i2c_read_blocking(i2c_port, device_addr, buf, 8, false);
00101    if (ret != 8) {
00102        uart_print("Failed to read from BME280.", VerbosityLevel::ERROR);
00103        return false;
00104    }
00105
00106     // Combine bytes to form raw values
00107     *pressure = ((int32_t)buf[0] << 12) | ((int32_t)buf[1] << 4) | ((int32_t)(buf[2] >> 4));
00108     *temperature = ((int32_t)buf[3] << 12) | ((int32_t)buf[4] << 4) | ((int32_t)(buf[5] >> 4));
00109     *humidity = ((int32_t)buf[6] << 8) | (int32_t)buf[7];
00110
00111     return true;
00112 }
00113
00119 float BME280::convert_temperature(int32_t temp_raw) const {
00120     int32_t var1 = (((temp_raw >> 3) - ((int32_t)calib_params.dig_t1 << 1))) * ((int32_t)calib_params.dig_t2)
00121     » 11;
00122     var2 = (((((temp_raw >> 4) - ((int32_t)calib_params.dig_t1)) * ((temp_raw >> 4) -
00123     ((int32_t)calib_params.dig_t1)) » 12) * ((int32_t)calib_params.dig_t3)) » 14;
00124     t_fine = var1 + var2;
00125     float T = (t_fine * 5 + 128) » 8;
00126     return T / 100.0f;
00127 }
00133 float BME280::convert_pressure(int32_t pressure_raw) const {
00134     int64_t var1, var2, p;
00135     var1 = ((int64_t)t_fine) - 128000;
00136     var2 = var1 * var1 * (int64_t)calib_params.dig_p6;
00137     var2 = var2 + ((var1 * (int64_t)calib_params.dig_p5) » 17);
00138     var2 = var2 + (((int64_t)calib_params.dig_p4) » 35);
00139     var1 = ((var1 * var1 * (int64_t)calib_params.dig_p3) » 8) + ((var1 * (int64_t)calib_params.dig_p2)
00140     » 12);
00140     var1 = (((int64_t)1 << 47) + var1) * ((int64_t)calib_params.dig_p1) » 33;
00141
00142     if (var1 == 0) {
00143         return 0.0f; // avoid exception caused by division by zero
00144     }
00145     p = 1048576 - pressure_raw;
00146     p = (((p << 31) - var2) * 3125) / var1;
00147     var1 = (((int64_t)calib_params.dig_p9) * (p » 13) * (p » 13)) » 25;
00148     var2 = (((int64_t)calib_params.dig_p8) * p) » 19;

```

```

00149
00150     p = ((p + var1 + var2) >> 8) + (((int64_t)calib_params.dig_p7) << 4);
00151     return (float)p / 25600.0f; // in hPa
00152 }
00153
00154 float BME280::convert_humidity(int32_t humidity_raw) const {
00155     int32_t v_x1_u32r;
00156     v_x1_u32r = t_fine - 76800;
00157     v_x1_u32r = (((humidity_raw << 14) - ((int32_t)calib_params.dig_h4 << 20) -
00158     ((int32_t)calib_params.dig_h5 * v_x1_u32r)) + 16384) >> 15) *
00159     (((((v_x1_u32r * (int32_t)calib_params.dig_h6) >> 10) * (((v_x1_u32r *
00160     (int32_t)calib_params.dig_h3) >> 11) + 32768)) >> 10) + 2097152) *
00161     (int32_t)calib_params.dig_h2 + 8192) >> 14));
00162     v_x1_u32r = std::max(v_x1_u32r, (int32_t)0);
00163     v_x1_u32r = std::min(v_x1_u32r, (int32_t)419430400);
00164     float h = v_x1_u32r >> 12;
00165     return h / 1024.0f;
00166 }
00167
00168 bool BME280::get_calibration_parameters() {
00169     // Read temperature and pressure calibration data (0x88 to 0xA1)
00170     uint8_t calib_data[NUM_CALIB_PARAMS];
00171     if (!read_register(REG_DIG_T1_LSB, calib_data, NUM_CALIB_PARAMS)) {
00172         uart_print("Failed to read calibration data from BME280.", VerbosityLevel::ERROR);
00173         return false;
00174     }
00175
00176     // Parse temperature calibration data
00177     calib_params.dig_t1 = (uint16_t)(calib_data[1] << 8 | calib_data[0]);
00178     calib_params.dig_t2 = (int16_t)(calib_data[3] << 8 | calib_data[2]);
00179     calib_params.dig_t3 = (int16_t)(calib_data[5] << 8 | calib_data[4]);
00180
00181     // Parse pressure calibration data
00182     calib_params.dig_p1 = (uint16_t)(calib_data[7] << 8 | calib_data[6]);
00183     calib_params.dig_p2 = (int16_t)(calib_data[9] << 8 | calib_data[8]);
00184     calib_params.dig_p3 = (int16_t)(calib_data[11] << 8 | calib_data[10]);
00185     calib_params.dig_p4 = (int16_t)(calib_data[13] << 8 | calib_data[12]);
00186     calib_params.dig_p5 = (int16_t)(calib_data[15] << 8 | calib_data[14]);
00187     calib_params.dig_p6 = (int16_t)(calib_data[17] << 8 | calib_data[16]);
00188     calib_params.dig_p7 = (int16_t)(calib_data[19] << 8 | calib_data[18]);
00189     calib_params.dig_p8 = (int16_t)(calib_data[21] << 8 | calib_data[20]);
00190     calib_params.dig_p9 = (int16_t)(calib_data[23] << 8 | calib_data[22]);
00191
00192     calib_params.dig_h1 = calib_data[25];
00193
00194     // Read humidity calibration data (0xE1 to 0xE7)
00195     uint8_t hum_calib_data[NUM_HUM_CALIB_PARAMS];
00196     if (!read_register(REG_DIG_H2, hum_calib_data, NUM_HUM_CALIB_PARAMS)) {
00197         uart_print("Failed to read humidity calibration data from BME280.", VerbosityLevel::ERROR);
00198         return false;
00199     }
00200
00201     // Parse humidity calibration data
00202     calib_params.dig_h2 = (int16_t)(hum_calib_data[1] << 8 | hum_calib_data[0]);
00203     calib_params.dig_h3 = hum_calib_data[2];
00204     calib_params.dig_h4 = (int16_t)((hum_calib_data[3] << 4) | (hum_calib_data[4] & 0x0F));
00205     calib_params.dig_h5 = (int16_t)((hum_calib_data[5] << 4) | (hum_calib_data[4] >> 4));
00206     calib_params.dig_h6 = (int8_t)hum_calib_data[6];
00207
00208     return true;
00209 }
00210
00211 bool BME280::configure_sensor() {
00212     // Set humidity oversampling (must be set before ctrl_meas)
00213     if (!write_register(REG_CTRL_HUM, HUMIDITY_OVERSAMPLING)) {
00214         uart_print("Failed to write CTRL_HUM to BME280.", VerbosityLevel::ERROR);
00215         return false;
00216     }
00217
00218     // Write config register
00219     if (!write_register(REG_CONFIG, 0x00)) {
00220         uart_print("Failed to write CONFIG to BME280.", VerbosityLevel::ERROR);
00221         return false;
00222     }
00223
00224     // Write ctrl_meas register
00225     if (!write_register(REG_CTRL_MEAS, NORMAL_MODE)) {
00226         uart_print("Failed to write CTRL_MEAS to BME280.", VerbosityLevel::ERROR);
00227         return false;
00228     }
00229
00230     return true;
00231 }
00232
00233 bool BME280::write_register(uint8_t reg, uint8_t value) {
00234     uint8_t buf[2] = {reg, value};
00235     int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
00236 }
```

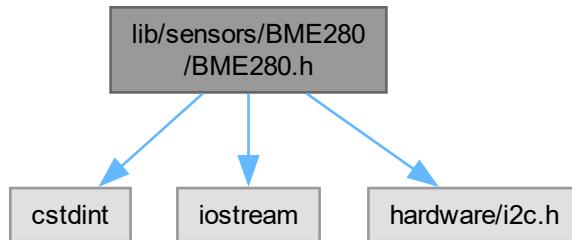
```
00253     return (ret == 2);
00254 }
00255
00263 bool BME280::read_register(uint8_t reg, uint8_t* data, size_t len) {
00264     int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
00265     if (ret != 1) {
00266         return false;
00267     }
00268     ret = i2c_read_blocking(i2c_port, device_addr, data, len, false);
00269     return (static_cast<size_t>(ret) == len);
00270 }
00271
00278 bool BME280::read_register(uint8_t reg, uint8_t* data) {
00279     return read_register(reg, data, 1);
00280 }
```

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

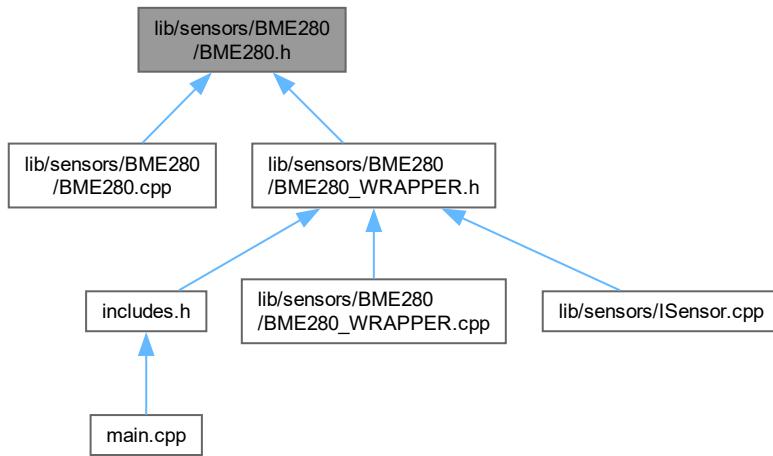
Header file for the [BME280](#) environmental sensor class.

```
#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](#)
Structure to hold the [BME280](#) calibration parameters.
- class [BME280](#)
Class to interface with the [BME280](#) environmental sensor.

9.68.1 Detailed Description

Header file for the [BME280](#) environmental sensor class.

This class provides an interface to the [BME280](#) temperature, pressure, and humidity sensor using the I2C communication protocol. It includes functions for initialization, reading raw sensor data, converting raw data to physical units, and configuring the sensor's operating mode.

Definition in file [BME280.h](#).

9.69 BME280.h

[Go to the documentation of this file.](#)

```

00001
00009
00010 #ifndef BME280_H
00011 #define BME280_H
00012
00013 #include <cstdint>
00014 #include <iostream>
00015 #include "hardware/i2c.h"
00016
00023 struct BME280CalibParam {
00025     uint16_t dig_t1;
00027     int16_t  dig_t2;
00029     int16_t  dig_t3;
  
```

```
00030
00032     uint16_t dig_p1;
00034     int16_t dig_p2;
00036     int16_t dig_p3;
00038     int16_t dig_p4;
00040     int16_t dig_p5;
00042     int16_t dig_p6;
00044     int16_t dig_p7;
00046     int16_t dig_p8;
00048     int16_t dig_p9;
00049
00050
00051     uint8_t dig_h1;
00053     int16_t dig_h2;
00055     uint8_t dig_h3;
00057     int16_t dig_h4;
00059     int16_t dig_h5;
00061     int8_t dig_h6;
00062 };
00063
00071 class BME280 {
00072 public:
00076     enum {
00078         ADDR_SDO_LOW = 0x76,
00080         ADDR_SDO_HIGH = 0x77
00081     };
00082
00089     enum class Oversampling : uint8_t {
00091         OSR_X0 = 0x00,
00093         OSR_X1 = 0x01,
00095         OSR_X2 = 0x02,
00097         OSR_X4 = 0x03,
00099         OSR_X8 = 0x04,
00101         OSR_X16 = 0x05
00102     };
00103
00109     BME280(i2c_inst_t* i2cPort, uint8_t address = ADDR_SDO_LOW);
00110
00115     bool init();
00116
00120     void reset();
00121
00129     bool read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity);
00130
00136     float convert_temperature(int32_t temp_raw) const;
00137
00143     float convert_pressure(int32_t pressure_raw) const;
00144
00150     float convert_humidity(int32_t humidity_raw) const;
00151
00152 private:
00159     bool write_register(uint8_t reg, uint8_t value);
00160
00167     bool read_register(uint8_t reg, uint8_t* data);
00168
00176     bool read_register(uint8_t reg, uint8_t* data, size_t len);
00177
00182     bool configure_sensor();
00183
00188     bool get_calibration_parameters();
00189
00191     i2c_inst_t* i2c_port;
00193     uint8_t device_addr;
00194
00196     BME280CalibParam calib_params;
00197
00199     bool initialized_;
00200
00202     mutable int32_t t_fine;
00203
00204     enum {
00208         REG_CONFIG          = 0xF5,
00209         REG_CTRL_MEAS       = 0xF4,
00210         REG_CTRL_HUM        = 0xF2,
00211         REG_RESET           = 0xE0,
00212
00213         REG_PRESSURE_MSB    = 0xF7,
00214         REG_TEMPERATURE_MSB = 0xFA,
00215         REG_HUMIDITY_MSB    = 0xFD,
00216
00217         // Calibration Registers
00218         REG_DIG_T1_LSB      = 0x88,
00219         REG_DIG_T1_MSB      = 0x89,
00220         REG_DIG_T2_LSB      = 0x8A,
00221         REG_DIG_T2_MSB      = 0x8B,
00222         REG_DIG_T3_LSB      = 0x8C,
00223         REG_DIG_T3_MSB      = 0x8D,
```

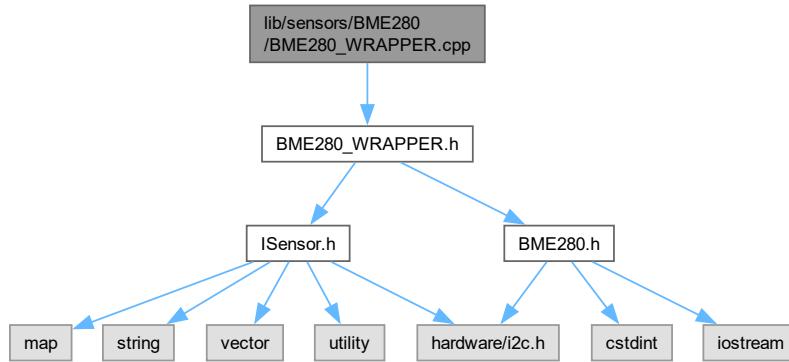
```

00225     REG_DIG_P1_LSB      = 0x8E,
00226     REG_DIG_P1_MSB      = 0x8F,
00227     REG_DIG_P2_LSB      = 0x90,
00228     REG_DIG_P2_MSB      = 0x91,
00229     REG_DIG_P3_LSB      = 0x92,
00230     REG_DIG_P3_MSB      = 0x93,
00231     REG_DIG_P4_LSB      = 0x94,
00232     REG_DIG_P4_MSB      = 0x95,
00233     REG_DIG_P5_LSB      = 0x96,
00234     REG_DIG_P5_MSB      = 0x97,
00235     REG_DIG_P6_LSB      = 0x98,
00236     REG_DIG_P6_MSB      = 0x99,
00237     REG_DIG_P7_LSB      = 0x9A,
00238     REG_DIG_P7_MSB      = 0x9B,
00239     REG_DIG_P8_LSB      = 0x9C,
00240     REG_DIG_P8_MSB      = 0x9D,
00241     REG_DIG_P9_LSB      = 0x9E,
00242     REG_DIG_P9_MSB      = 0x9F,
00243
00244 // Humidity Calibration Registers
00245     REG_DIG_H1          = 0xA1,
00246     REG_DIG_H2          = 0xE1,
00247     REG_DIG_H3          = 0xE3,
00248     REG_DIG_H4          = 0xE4,
00249     REG_DIG_H5          = 0xE5,
00250     REG_DIG_H6          = 0xE7
00251 };
00252
00253 enum {
00254     HUMIDITY_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
00255     TEMPERATURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
00256     PRESSURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
00257     NORMAL_MODE = 0xB7
00258 };
00259
00260 enum {
00261     NUM_CALIB_PARAMS = 26,
00262     NUM_HUM_CALIB_PARAMS = 7
00263 };
00264
00265 };
00266
00267 #endif // BME280_H

```

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

```
#include "BME280_WRAPPER.h"
Include dependency graph for BME280_WRAPPER.cpp:
```



9.71 BME280_WRAPPER.cpp

[Go to the documentation of this file.](#)

```

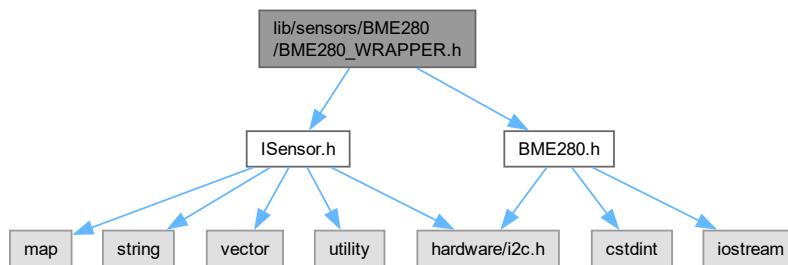
00001 #include "BME280_WRAPPER.h"
00002
00003 BME280Wrapper::BME280Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {}
00004
00005 bool BME280Wrapper::init() {
00006     initialized_ = sensor_.init();
00007     return initialized_;
00008 }
00009
00010 float BME280Wrapper::read_data(SensorDataTypeIdentifier type) {
00011     int32_t temp_raw, pressure_raw, humidity_raw;
00012     sensor_.read_raw_all(&temp_raw, &pressure_raw, &humidity_raw);
00013
00014     switch(type) {
00015         case SensorDataTypeIdentifier::TEMPERATURE:
00016             return sensor_.convert_temperature(temp_raw);
00017         case SensorDataTypeIdentifier::PRESSURE:
00018             return sensor_.convert_pressure(pressure_raw);
00019         case SensorDataTypeIdentifier::HUMIDITY:
00020             return sensor_.convert_humidity(humidity_raw);
00021         default:
00022             return 0.0f;
00023     }
00024 }
00025
00026 bool BME280Wrapper::is_initialized() const {
00027     return initialized_;
00028 }
00029
00030 SensorType BME280Wrapper::get_type() const {
00031     return SensorType::ENVIRONMENT;
00032 }
00033
00034 bool BME280Wrapper::configure([[maybe_unused]] const std::map<std::string, std::string>& config) {
00035     return true;
00036 }

```

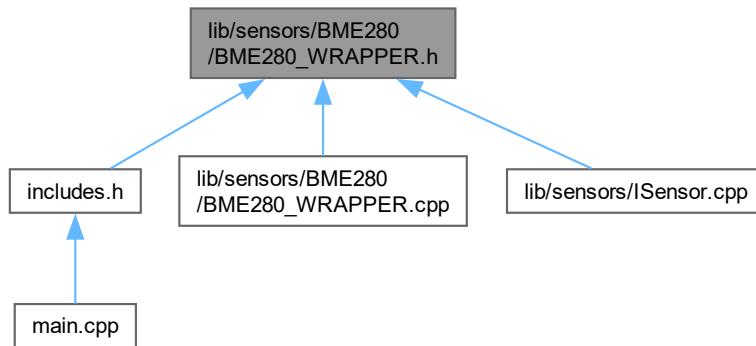
9.72 lib/sensors/BME280/BME280_WRAPPER.h File Reference

```
#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](#)

9.73 BME280_WRAPPER.h

[Go to the documentation of this file.](#)

```

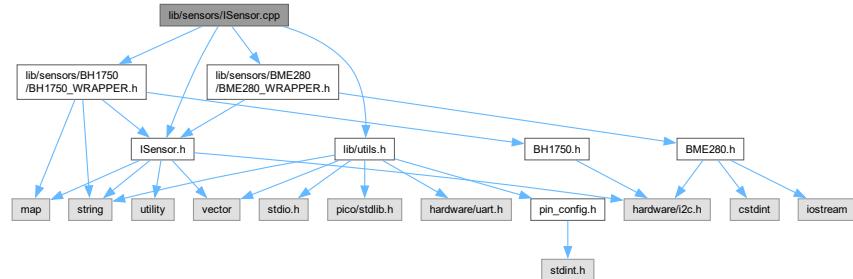
00001 // BME280_WRAPPER.h
00002 #ifndef BME280_WRAPPER_H
00003 #define BME280_WRAPPER_H
00004
00005 #include "ISensor.h"
00006 #include "BME280.h"
00007
00008 class BME280Wrapper : public ISensor {
00009 private:
00010     BME280 sensor_;
00011     bool initialized_ = false;
00012
00013 public:
00014     BME280Wrapper(i2c_inst_t* i2c);
00015
00016     bool init() override;
00017     float read_data(SensorDataTypeIdentifier type) override;
00018     bool is_initialized() const override;
00019     SensorType get_type() const override;
00020     bool configure(const std::map<std::string, std::string>& config) override;
00021
00022     uint8_t get_address() const override {
00023         return 0x76;
00024     }
00025
00026 };
00027
00028 #endif // BME280_WRAPPER_H
  
```

9.74 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:
```



9.74.1 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](#).

9.75 ISensor.cpp

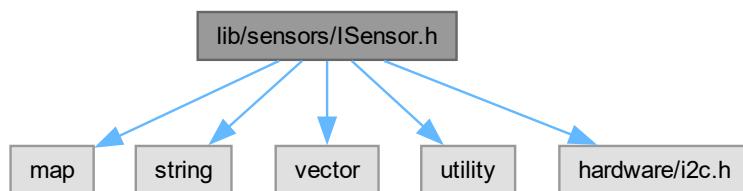
[Go to the documentation of this file.](#)

```
00001
00014
00015 #include "ISensor.h"
00016 #include "lib/sensors/BH1750/BH1750_WRAPPER.h"
00017 #include "lib/sensors/BME280/BME280_WRAPPER.h"
00018 #include "lib/utils.h"
00019
00027 bool SensorWrapper::sensor_init(SensorType type, i2c_inst_t* i2c) {
00028     switch (type) {
00029         case SensorType::LIGHT:
00030             sensors[type] = new BH1750Wrapper(i2c);
00031             break;
00032         case SensorType::ENVIRONMENT:
00033             sensors[type] = new BME280Wrapper(i2c);
00034             break;
00035         default:
00036             return false;
00037     }
00038     return sensors[type]->init();
00039 }
00040
00048 bool SensorWrapper::sensor_configure(SensorType type, const std::map<std::string, std::string>&
config) {
00049     if (sensors.find(type) == sensors.end()) {
00050         return false;
00051     }
00052     return sensors[type]->configure(config);
00053 }
00054
00062 float SensorWrapper::sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType) {
00063     if (sensors.find(sensorType) == sensors.end()) {
00064         return -1.0f;
00065     }
00066     return sensors[sensorType]->read_data(dataType);
00067 }
00068
00075 ISensor* SensorWrapper::get_sensor(SensorType type) {
00076     return sensors[type];
00077 }
```

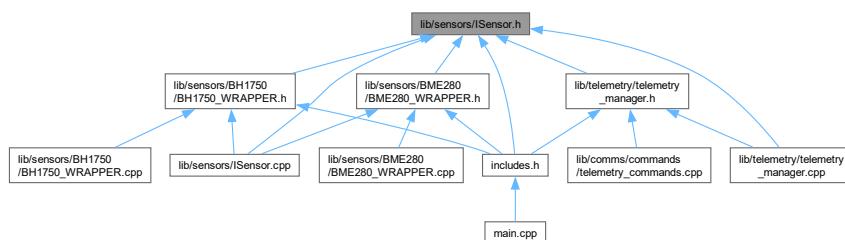
9.76 lib/sensors/ISensor.h File Reference

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 [SensorDataTypelIdentifier](#) : `uint8_t` {
`SensorDataTypelIdentifier::NONE` = `0x00` , `SensorDataTypelIdentifier::LIGHT_LEVEL` = `0x01` , `SensorDataTypelIdentifier::TEMPERATURE` = `0x02` , `SensorDataTypelIdentifier::HUMIDITY` = `0x03` ,
`SensorDataTypelIdentifier::PRESSURE` = `0x04` }
Enumeration of sensor data type identifiers.

9.76.1 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](#).

9.77 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;
```

```

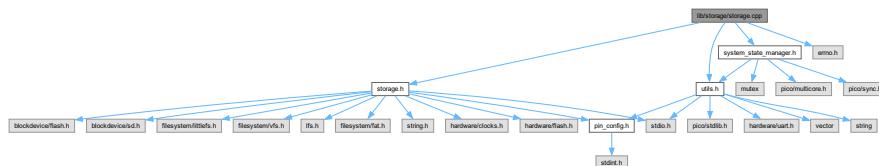
00174     SensorWrapper() = default;
00175 };
00176
00177 #endif

```

9.78 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.

9.78.1 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](#).

9.79 storage.cpp

[Go to the documentation of this file.](#)

```

00001
00012
00013 #include "storage.h"
00014 #include "errno.h"
00015 #include "utils.h"
00016 #include "system_state_manager.h"
00017
00025 bool fs_init(void) {
00026     SystemStateManager::get_instance().set_sd_card_mounted(false);
00027     uart_print("fs_init littlefs on SD card", VerbosityLevel::DEBUG);
00028     blockdevice_t *sd = blockdevice_sd_create(SD_SPI_PORT,
00029                                              SD_MOSI_PIN,

```

```

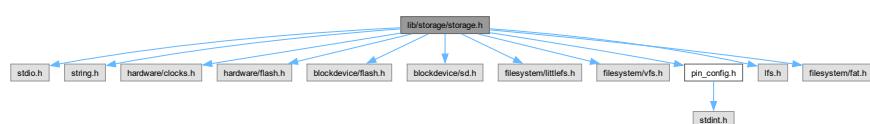
00030                               SD_MISO_PIN,
00031                               SD_SCK_PIN,
00032                               SD_CS_PIN,
00033                               24 * MHZ,
00034                               false);
00035     filesystem_t *fat = filesystem_fat_create();
00036
00037     std::string status_string;
00038     int err = fs_mount("/", fat, sd);
00039     if (err == -1) {
00040         status_string = "Formatting / with FAT";
00041         uart_print(status_string, VerbosityLevel::WARNING);
00042         err = fs_format(fat, sd);
00043         if (err == -1) {
00044             status_string = "fs_format error: " + std::string(strerror(errno));
00045             uart_print(status_string, VerbosityLevel::ERROR);
00046             return false;
00047         }
00048         err = fs_mount("/", fat, sd);
00049         if (err == -1) {
00050             status_string = "fs_mount error: " + std::string(strerror(errno));
00051             uart_print(status_string, VerbosityLevel::ERROR);
00052             return false;
00053         }
00054     }
00055
00056     SystemStateManager::get_instance().set_sd_card_mounted(true);
00057     return true;
00058 }
00059
00060 bool fs_stop(void) {
00061     int err = fs_unmount("/");
00062     if (err == -1) {
00063         uart_print("fs_unmount error", VerbosityLevel::ERROR);
00064         return false;
00065     }
00066     SystemStateManager::get_instance().set_sd_card_mounted(false);
00067     return true;
00068 }
```

9.80 lib/storage/storage.h File Reference

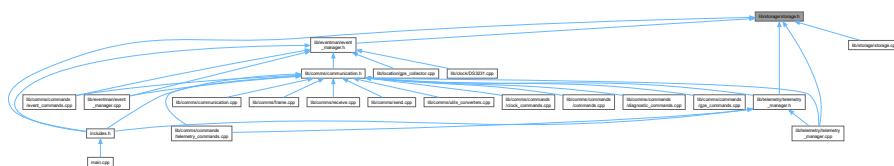
Header file for file system operations on the Kubisat firmware.

```
#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.

9.80.1 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](#).

9.81 storage.h

[Go to the documentation of this file.](#)

```

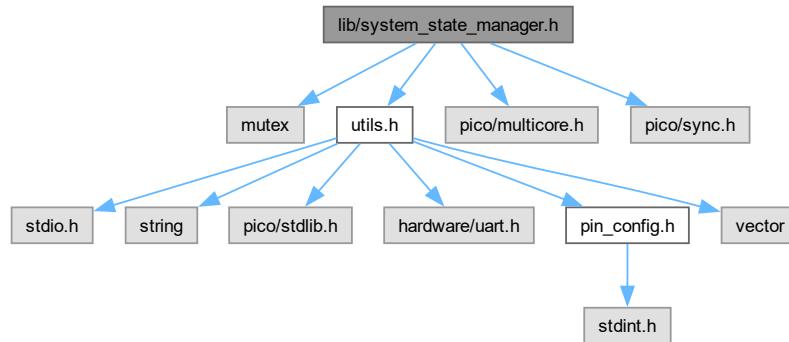
00001
00013
00014 #ifndef STORAGE_H
00015 #define STORAGE_H
00016
00017 #include <stdio.h>
00018 #include <string.h>
00019 #include <hardware/clocks.h>
00020 #include <hardware/flash.h>
00021 #include "blockdevice/flash.h"
00022 #include "blockdevice/sd.h"
00023 #include "filesystem/littlefs.h"
00024 #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
  
```

9.82 lib/system_state_manager.h File Reference

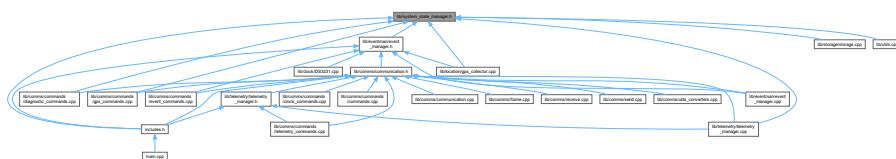
Manages the system state of the Kabisat firmware.

```
#include <mutex>
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
Include dependency graph for system_state_manager.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.

Enumerations

- enum class `SystemOperatingMode` : `uint8_t` { `SystemOperatingMode::BATTERY_POWERED` = 0 ,
`SystemOperatingMode::USB_POWERED` = 1 }

Enumeration of system operating modes.

9.82.1 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](#).

9.83 system_state_manager.h

[Go to the documentation of this file.](#)

```

00001
00015
00016 #ifndef SYSTEM_STATE_MANAGER_H
00017 #define SYSTEM_STATE_MANAGER_H
00018
00019 #include <mutex>
00020 #include "utils.h"
00021 #include "pico/multicore.h"
00022 #include "pico/sync.h"
00023
00027 enum class SystemOperatingMode : uint8_t {
00028     BATTERY_POWERED = 0,
00029     USB_POWERED = 1
00030 };
00031
00040 class SystemStateManager {
00041     private:
00043         bool pending_bootloader_reset;
00045         bool gps_collection_paused;
00047         bool sd_card_mounted;
00049         VerbosityLevel uart_verbosity;
00051         bool sd_card_init_status;
00053         bool radio_init_status;
00055         bool light_sensor_init_status;
00057         bool env_sensor_init_status;
00059         SystemOperatingMode system_operating_mode;
00061         recursive_mutex_t mutex_;
00062
00067     SystemStateManager() :
00068         pending_bootloader_reset(false),
00069         gps_collection_paused(false),
00070         sd_card_mounted(false),
00071         uart_verbosity(VerbosityLevel::DEBUG),
00072         sd_card_init_status(false),
00073         radio_init_status(false),
00074         light_sensor_init_status(false),
00075         env_sensor_init_status(false),
00076         system_operating_mode(SystemOperatingMode::BATTERY_POWERED)
00077     {
00078         recursive_mutex_init(&mutex_);
00079     }
00080
00084     SystemStateManager(const SystemStateManager&) = delete;
00085     SystemStateManager& operator=(const SystemStateManager&) = delete;
00089
00090     public:
00095         static SystemStateManager& get_instance() {
00096             static SystemStateManager instance;
00097             return instance;
00098         }
00099
00104         bool is_bootloader_reset_pending() const {
00105             recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00106             bool result = pending_bootloader_reset;
00107             recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00108             return result;
00109         }
00110
00115         void set_bootloader_reset_pending(bool pending) {
00116             recursive_mutex_enter_blocking(&mutex_);
00117             pending_bootloader_reset = pending;
00118             recursive_mutex_exit(&mutex_);
00119         }
00120
00125         bool is_gps_collection_paused() const {
00126             recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00127             bool result = gps_collection_paused;
00128             recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00129             return result;
00130         }
00131
00136         void set_gps_collection_paused(bool paused) {
00137             recursive_mutex_enter_blocking(&mutex_);
00138             gps_collection_paused = paused;
00139             recursive_mutex_exit(&mutex_);
00140         }
00141
00146         bool is_sd_card_mounted() const {
00147             recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00148             bool result = sd_card_mounted;
00149             recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00150             return result;

```

```

00151         }
00152
00153     void set_sd_card_mounted(bool mounted) {
00154         recursive_mutex_enter_blocking(&mutex_);
00155         sd_card_mounted = mounted;
00156         recursive_mutex_exit(&mutex_);
00157     }
00158
00159     VerbosityLevel get_uart_verbosity() const {
00160         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00161         VerbosityLevel result = uart_verbosity;
00162         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00163         return result;
00164     }
00165
00166     void set_uart_verbosity(VerbosityLevel level) {
00167         recursive_mutex_enter_blocking(&mutex_);
00168         uart_verbosity = level;
00169         recursive_mutex_exit(&mutex_);
00170     }
00171
00172     bool is_radio_init_ok() const {
00173         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00174         bool result = radio_init_status;
00175         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00176         return result;
00177     }
00178
00179     void set_radio_init_ok(bool status) {
00180         recursive_mutex_enter_blocking(&mutex_);
00181         radio_init_status = status;
00182         recursive_mutex_exit(&mutex_);
00183     }
00184
00185     bool is_light_sensor_init_ok() const {
00186         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00187         bool result = light_sensor_init_status;
00188         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00189         return result;
00190     }
00191
00192     void set_light_sensor_init_ok(bool status) {
00193         recursive_mutex_enter_blocking(&mutex_);
00194         light_sensor_init_status = status;
00195         recursive_mutex_exit(&mutex_);
00196     }
00197
00198     bool is_env_sensor_init_ok() const {
00199         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00200         bool result = env_sensor_init_status;
00201         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00202         return result;
00203     }
00204
00205     SystemOperatingMode get_operating_mode() const {
00206         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00207         SystemOperatingMode result = system_operating_mode;
00208         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00209         return result;
00210     }
00211
00212     void set_operating_mode(SystemOperatingMode mode) {
00213         recursive_mutex_enter_blocking(&mutex_);
00214         system_operating_mode = mode;
00215         recursive_mutex_exit(&mutex_);
00216     }
00217 }
00218
00219 #endif

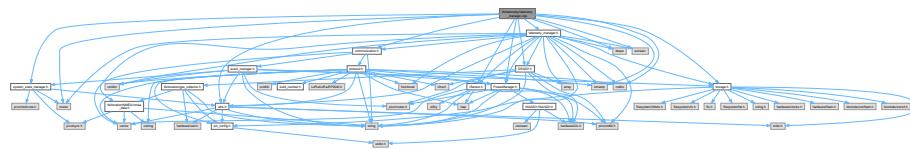
```

9.84 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.

9.84.1 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](#).

9.85 `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
00044 TelemetryManager::TelemetryManager() {}
00045
00054 bool TelemetryManager::init() {
00055     mutex_init(&telemetry_mutex);
00056     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00057         uart_print("Telemetry system initialized (storage not available)", VerbosityLevel::WARNING);
00058         return false;
00059     }
00060
00061     bool success = true;
00062
00063     FILE* telemetry_file = fopen(TELEMETRY_CSV_PATH, "w");
00064     if (!telemetry_file) {
00065         telemetry_file = fopen(TELEMETRY_CSV_PATH, "w");
00066         if (telemetry_file) {
00067             fprintf(telemetry_file, "timestamp,build,battery_v,system_v,usb_ma,solar_ma,discharge_ma,"
00068                     "gps_time,latitude,lat_dir,longitude,lon_dir,speed_mps,course_deg,date,"
00069                     "fix_quality,satellites,altitude_m\n");
00070             fclose(telemetry_file);
00071             uart_print("Created new telemetry log", VerbosityLevel::INFO);
00072         }
00073         else {
00074             uart_print("Failed to create telemetry log", VerbosityLevel::ERROR);
00075             success = false;
00076         }
00077     }
00078     else {
00079         fclose(telemetry_file);
00080     }
00081
00082     FILE* sensor_file = fopen(SENSOR_DATA_CSV_PATH, "w");
00083     if (!sensor_file) {
00084         sensor_file = fopen(SENSOR_DATA_CSV_PATH, "w");
00085         if (sensor_file) {
00086             fprintf(sensor_file, "timestamp,temperature,pressure,humidity,light\n");
00087             fclose(sensor_file);
00088             uart_print("Created new sensor data log", VerbosityLevel::INFO);
00089         }
00090         else {
00091             uart_print("Failed to create sensor data log", VerbosityLevel::ERROR);
00092             success = false;
00093         }
00094     }
00095     else {
00096         fclose(sensor_file);
00097     }
00098
00099     return success;
00100 }
00101
00102
00108 void TelemetryManager::collect_power_telemetry(TelemetryRecord& record) {
00109     record.battery_voltage = PowerManager::get_instance().get_voltage_battery();
00110     record.system_voltage = PowerManager::get_instance().get_voltage_5v();
00111     record.charge_current_usb = PowerManager::get_instance().get_current_charge_usb();
00112     record.charge_current_solar = PowerManager::get_instance().get_current_charge_solar();
00113     record.discharge_current = PowerManager::get_instance().get_current_draw();
00114 }
00115
00124 void TelemetryManager::emit_power_events(float battery_voltage, float charge_current_usb, float
charge_current_solar, float discharge_current) {
00125     static bool usb_charging_active = false;
00126     static bool solar_charging_active = false;
00127     static bool battery_low = false;
00128     static bool battery_full = false;
00129     static bool discharge_active = true;
00130
00131     if (charge_current_usb > PowerManager::USB_CURRENT_THRESHOLD && !usb_charging_active) {
00132         EventEmitter::emit(EventGroup::POWER, PowerEvent::USB_CONNECTED);
00133         usb_charging_active = true;
00134     }
00135     else if (charge_current_usb < PowerManager::USB_CURRENT_THRESHOLD && usb_charging_active) {
00136         EventEmitter::emit(EventGroup::POWER, PowerEvent::USB_DISCONNECTED);
00137         usb_charging_active = false;
00138     }

```

```

00139 if (charge_current_solar > PowerManager::SOLAR_CURRENT_THRESHOLD && !solar_charging_active) {
00140     EventEmitter::emit(EventGroup::POWER, PowerEvent::SOLAR_ACTIVE);
00141     solar_charging_active = true;
00142 }
00143 else if (charge_current_solar < PowerManager::SOLAR_CURRENT_THRESHOLD && solar_charging_active) {
00144     EventEmitter::emit(EventGroup::POWER, PowerEvent::SOLAR_INACTIVE);
00145     solar_charging_active = false;
00146 }
00147 }
00148
00149 if (battery_voltage < PowerManager::BATTERY_LOW_THRESHOLD && !battery_low) {
00150     EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_LOW);
00151     battery_low = true;
00152     battery_full = false;
00153 }
00154 else if (battery_voltage > PowerManager::BATTERY_FULL_THRESHOLD && !battery_full) {
00155     EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_FULL);
00156     battery_full = true;
00157     battery_low = false;
00158 }
00159 else if (battery_voltage > PowerManager::BATTERY_LOW_THRESHOLD && battery_low) {
00160     EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_NORMAL);
00161     battery_low = false;
00162 }
00163 else if (battery_voltage < PowerManager::BATTERY_FULL_THRESHOLD && battery_full) {
00164     EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_NORMAL);
00165     battery_full = false;
00166 }
00167
00168 if (charge_current_solar + charge_current_usb > discharge_current && !discharge_active) {
00169     EventEmitter::emit(EventGroup::POWER, PowerEvent::CHARGING);
00170     discharge_active = true;
00171 }
00172 else if (charge_current_solar + charge_current_usb < discharge_current && discharge_active) {
00173     EventEmitter::emit(EventGroup::POWER, PowerEvent::DISCHARGING);
00174     discharge_active = false;
00175 }
00176 }
00177
00183 void TelemetryManager::collect_gps_telemetry(TelemetryRecord& record) {
00184     auto& nmea_data = NMEAData::get_instance();
00185
00186     std::vector<std::string> rmc_tokens = nmea_data.get_rmc_tokens();
00187     if (rmc_tokens.size() >= 12) {
00188         record.time = (rmc_tokens.size() > 1 && !rmc_tokens[1].empty()) ? rmc_tokens[1].substr(0, 6) :
00189             "0"; // Only keep HHMMSS
00190         record.latitude = (rmc_tokens.size() > 3 && !rmc_tokens[3].empty()) ? rmc_tokens[3] : "0";
00191         record.lat_dir = (rmc_tokens.size() > 4 && !rmc_tokens[4].empty()) ? rmc_tokens[4] : "N";
00192         record.longitude = (rmc_tokens.size() > 5 && !rmc_tokens[5].empty()) ? rmc_tokens[5] : "0";
00193         record.lon_dir = (rmc_tokens.size() > 6 && !rmc_tokens[6].empty()) ? rmc_tokens[6] : "E";
00194
00195         std::string knots = (rmc_tokens.size() > 7 && !rmc_tokens[7].empty()) ? rmc_tokens[7] : "0";
00196         try {
00197             float speed_knots = std::stof(knots);
00198             float speed_ms = speed_knots * 0.514444; // Convert knots to m/s
00199             record.speed = std::to_string(speed_ms);
00200         } catch (const std::exception&) {
00201             record.speed = "0";
00202         }
00203
00204         record.course = (rmc_tokens.size() > 8 && !rmc_tokens[8].empty()) ? rmc_tokens[8] : "0";
00205         record.date = (rmc_tokens.size() > 9 && !rmc_tokens[9].empty()) ? rmc_tokens[9] : "0";
00206     } else {
00207         record.time = "0";
00208         record.latitude = "0";
00209         record.lat_dir = "N";
00210         record.longitude = "0";
00211         record.lon_dir = "E";
00212         record.speed = "0";
00213         record.course = "0";
00214         record.date = "0";
00215     }
00216
00217     std::vector<std::string> gga_tokens = nmea_data.get_gga_tokens();
00218     if (gga_tokens.size() >= 15) {
00219         record.fix_quality = (gga_tokens.size() > 6 && !gga_tokens[6].empty()) ? gga_tokens[6] : "0";
00220         record.satellites = (gga_tokens.size() > 7 && !gga_tokens[7].empty()) ? gga_tokens[7] : "0";
00221         record.altitude = (gga_tokens.size() > 9 && !gga_tokens[9].empty()) ? gga_tokens[9] : "0";
00222     } else {
00223         record.fix_quality = "0";
00224         record.satellites = "0";
00225         record.altitude = "0";
00226     }
00227
00233 void TelemetryManager::collect_sensor_telemetry(SensorDataRecord& sensor_record) {
00234     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();

```

```

00235     sensor_record.temperature = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00236     SensorDataTypeIdentifier::TEMPERATURE);
00237     sensor_record.pressure = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00238     SensorDataTypeIdentifier::PRESSURE);
00239     sensor_record.humidity = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00240     SensorDataTypeIdentifier::HUMIDITY);
00241     sensor_record.light = sensor_wrapper.sensor_read_data(SensorType::LIGHT,
00242     SensorDataTypeIdentifier::LIGHT_LEVEL);
00243 }
00244
00245 bool TelemetryManager::collect_telemetry() {
00246     uint32_t timestamp = DS3231::get_instance().get_local_time();
00247     TelemetryRecord record;
00248     record.timestamp = timestamp;
00249     record.build_version = std::to_string(BUILD_NUMBER);
00250
00251     collect_power_telemetry(record);
00252     emit_power_events(record.battery_voltage, record.charge_current_usb, record.charge_current_solar,
00253     record.discharge_current);
00254
00255     collect_gps_telemetry(record);
00256
00257     SensorDataRecord sensor_record;
00258     sensor_record.timestamp = timestamp;
00259     collect_sensor_telemetry(sensor_record);
00260
00261     mutex_enter_blocking(&telemetry_mutex);
00262
00263     telemetry_buffer[telemetry_buffer_write_index] = record;
00264     sensor_data_buffer[telemetry_buffer_write_index] = sensor_record;
00265     telemetry_buffer_write_index = (telemetry_buffer_write_index + 1) % TELEMETRY_BUFFER_SIZE;
00266     if (telemetry_buffer_count < TELEMETRY_BUFFER_SIZE) {
00267         telemetry_buffer_count++;
00268     }
00269 }
00270
00271     last_telemetry_record_copy = record;
00272     last_sensor_record_copy = sensor_record;
00273
00274     mutex_exit(&telemetry_mutex);
00275
00276     uart_print("Telemetry collected", VerbosityLevel::DEBUG);
00277
00278     return true;
00279 }
00280 }
00281
00282
00283 bool TelemetryManager::flush_telemetry() {
00284     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00285         return false;
00286     }
00287
00288     mutex_enter_blocking(&telemetry_mutex);
00289
00290     if (telemetry_buffer_count == 0) {
00291         mutex_exit(&telemetry_mutex);
00292         return true; // Nothing to save
00293     }
00294
00295     FILE* telemetry_file = fopen(TELEMETRY_CSV_PATH, "a");
00296     FILE* sensor_file = fopen(SENSOR_DATA_CSV_PATH, "a");
00297
00298     if (!telemetry_file || !sensor_file) {
00299         uart_print("Failed to open telemetry or sensor log for writing", VerbosityLevel::ERROR);
00300         if (telemetry_file) fclose(telemetry_file);
00301         if (sensor_file) fclose(sensor_file);
00302         mutex_exit(&telemetry_mutex);
00303         return false;
00304     }
00305
00306     // Calculate start index (for circular buffer)
00307     size_t read_index = 0;
00308     if (telemetry_buffer_count == TELEMETRY_BUFFER_SIZE) {
00309         // Buffer is full, start from oldest entry
00310         read_index = telemetry_buffer_write_index;
00311     }
00312
00313     // Write all records to CSV
00314     for (size_t i = 0; i < telemetry_buffer_count; i++) {
00315         fprintf(telemetry_file, "%s\n", telemetry_buffer[read_index].to_csv().c_str());
00316         fprintf(sensor_file, "%s\n", sensor_data_buffer[read_index].to_csv().c_str());
00317         read_index = (read_index + 1) % TELEMETRY_BUFFER_SIZE;
00318     }
00319
00320     // Clear buffer after successful write
00321     telemetry_buffer_count = 0;
00322     telemetry_buffer_write_index = 0;
00323 }
```

```

00331     fclose(telemetry_file);
00332     fclose(sensor_file);
00333
00334     mutex_exit(&telemetry_mutex);
00335     return true;
00336 }
00337
00346 bool TelemetryManager::is_telemetry_collection_time(uint32_t current_time, uint32_t&
last_collection_time) {
00347     if (current_time - last_collection_time >= sample_interval_ms) {
00348         last_collection_time = current_time;
00349         return true;
00350     }
00351     return false;
00352 }
00353
00354
00362 bool TelemetryManager::is_telemetry_flush_time(uint32_t& collection_counter) {
00363     if (collection_counter >= flush_threshold) {
00364         collection_counter = 0;
00365         return true;
00366     }
00367     return false;
00368 }
00369
00375 std::string TelemetryManager::get_last_telemetry_record_csv() {
00376     return last_telemetry_record_copy.to_csv();
00377 }
00378
00384 std::string TelemetryManager::get_last_sensor_record_csv() {
00385     return last_sensor_record_copy.to_csv();
00386 }

```

9.86 lib/telemetry/telemetry_manager.h File Reference

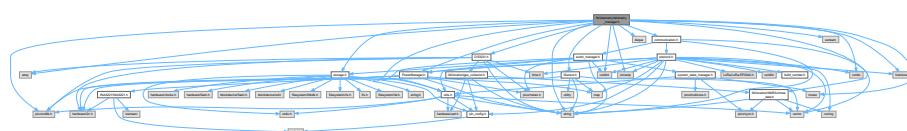
System telemetry collection and logging.

```

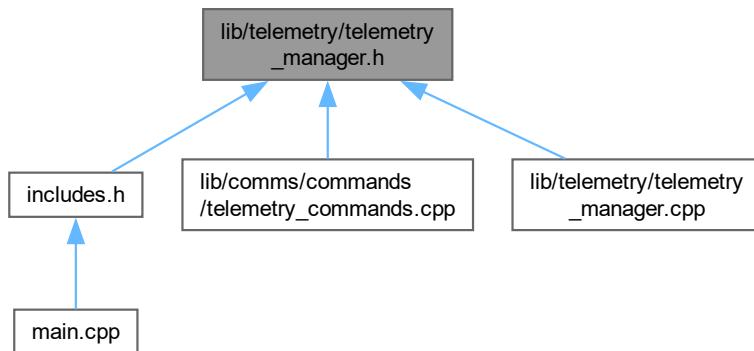
#include <cstdint>
#include <string>
#include "pico/stdlib.h"
#include "lib/location/NMEA/nmea_data.h"
#include "utils.h"
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include <array>
#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.
- class [TelemetryManager](#)
Manages the collection, storage, and retrieval of telemetry data.

9.86.1 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](#).

9.87 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 <array>
00035 #include "communication.h"
00036 #include <functional>
00037
00044 struct TelemetryRecord {
00045     uint32_t timestamp;
00046
00047     std::string build_version;
00048
00049     // Power data
00050     float battery_voltage;
00051     float system_voltage;
00052     float charge_current_usb;
00053     float charge_current_solar;
00054     float discharge_current;
00055
00056     // GPS data - key RMC fields
00057     std::string time;
00058     std::string latitude;
00059     std::string lat_dir;
00060     std::string longitude;
00061     std::string lon_dir;
00062     std::string speed;
00063     std::string course;
00064     std::string date;
00065
00066     // GPS data - key GGA fields
00067     std::string fix_quality;
00068     std::string satellites;
00069     std::string altitude;
00070
00071
00077     std::string to_csv() const {
00078         std::stringstream ss;
00079         ss << timestamp << ","
00080             << build_version << ","
00081             << std::fixed << std::setprecision(3)
00082             << battery_voltage << ","
00083             << system_voltage << ","
00084             << charge_current_usb << ","
00085             << charge_current_solar << ","
00086             << discharge_current << ","
00087
00088             // GPS RMC data
00089             << time << ","
00090             << latitude << "," << lat_dir << ","
00091             << longitude << "," << lon_dir << ","
00092             << speed << ","
00093             << course << ","
00094             << date << ","
00095
00096             // GPS GGA data
00097             << fix_quality << ","
00098             << satellites << ","
00099             << altitude;
00100     }
00101 };
00102
00103
00111 struct SensorDataRecord {
00112     uint32_t timestamp;
00113     float temperature;
00114     float pressure;
00115     float humidity;
00116     float light;
00117
00118     std::string to_csv() const {
00119         std::stringstream ss;
00120         ss << timestamp << ","
00121             << std::fixed << std::setprecision(3)
00122             << temperature << ","
00123             << pressure << ","
00124             << humidity << ","
00125             << light;
00126     }
00127 };
00128
00129
00130
00131
00132
00133 };
00134
```

```

00135
00146 class TelemetryManager {
00147 public:
00152     static TelemetryManager& get_instance() {
00153         static TelemetryManager instance;
00154         return instance;
00155     }
00156
00163     bool init();
00164
00171     bool collect_telemetry();
00172
00178     void collect_power_telemetry(TelemetryRecord& record);
00179
00188     void emit_power_events(float battery_voltage, float charge_current_usb, float
charge_current_solar, float discharge_current);
00189
00195     void collect_gps_telemetry(TelemetryRecord& record);
00196
00202     void collect_sensor_telemetry(SensorDataRecord& sensor_record);
00203
00210     bool flush_telemetry();
00211
00218     bool flush_sensor_data();
00219
00227     bool is_telemetry_collection_time(uint32_t current_time, uint32_t& last_collection_time);
00228
00235     bool is_telemetry_flush_time(uint32_t& collection_counter);
00236
00237
00242     std::string get_last_telemetry_record_csv();
00243
00248     std::string get_last_sensor_record_csv();
00249
00250     static constexpr int TELEMETRY_BUFFER_SIZE = 20;
00251
00252     size_t get_telemetry_buffer_count() const { return telemetry_buffer_count; }
00253     size_t get_telemetry_buffer_write_index() const { return telemetry_buffer_write_index; }
00254
00255 private:
00256     TelemetryManager(); // Private constructor
00257     ~TelemetryManager() = default;
00258
00262     static constexpr uint32_t DEFAULT_SAMPLE_INTERVAL_MS = 1000;
00263
00267     static constexpr uint32_t DEFAULT_FLUSH_THRESHOLD = 10;
00268
00269     uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS;
00270
00274     uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD;
00278     std::array<TelemetryRecord, TELEMETRY_BUFFER_SIZE> telemetry_buffer;
00279     size_t telemetry_buffer_count = 0;
00280     size_t telemetry_buffer_write_index = 0;
00281
00285     std::array<SensorDataRecord, TELEMETRY_BUFFER_SIZE> sensor_data_buffer;
00286
00290     TelemetryRecord last_telemetry_record_copy;
00291     SensorDataRecord last_sensor_record_copy;
00292
00296     mutex_t telemetry_mutex;
00297 };
00298 #endif // TELEMETRY_MANAGER_H
00299 // End of TelemetryManager group

```

9.88 lib/utils.cpp File Reference

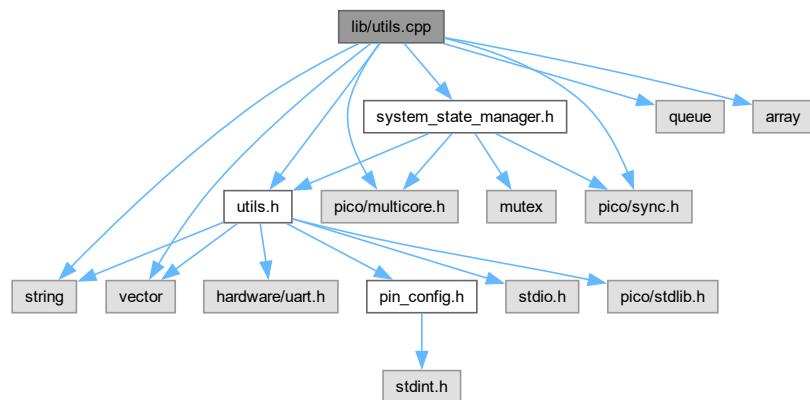
Implementation of utility functions for the KubitSat firmware.

```

#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.

9.88.1 Detailed Description

Implementation of utility functions for the Kubisat firmware.

Definition in file [utils.cpp](#).

9.88.2 Function Documentation

9.88.2.1 `get_level_color()`

```
std::string get_level_color (
    VerbosityLevel level)
```

Gets ANSI color code for verbosity level.

Parameters

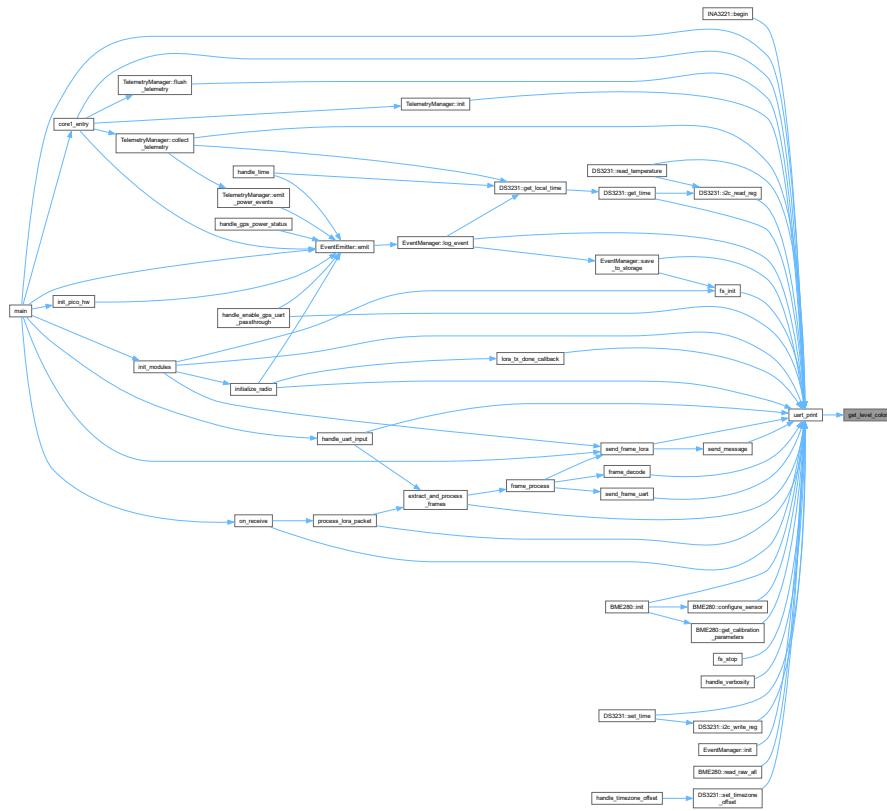
<code>level</code>	The verbosity level
--------------------	---------------------

Returns

ANSI color escape sequence

Definition at line 27 of file [utils.cpp](#).

Here is the caller graph for this function:



9.88.2.2 `get_level_prefix()`

```
std::string get_level_prefix (
    VerbosityLevel level)
```

Gets text prefix for verbosity level.

Parameters

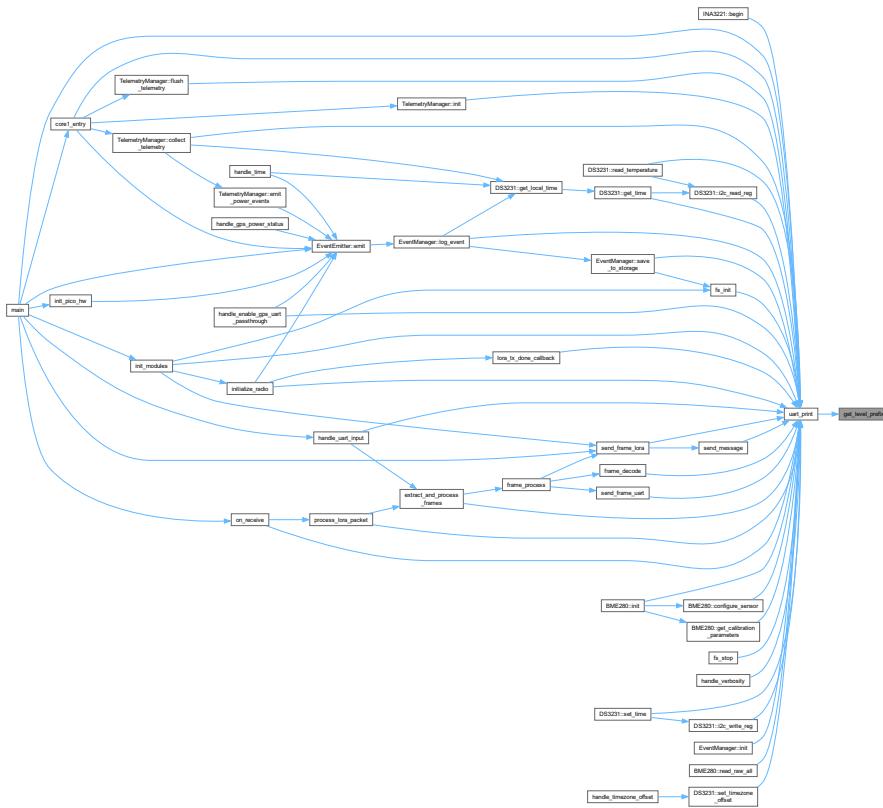
<code>level</code>	The verbosity level
--------------------	---------------------

Returns

Text prefix for the level

Definition at line 43 of file [utils.cpp](#).

Here is the caller graph for this function:



9.88.2.3 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.

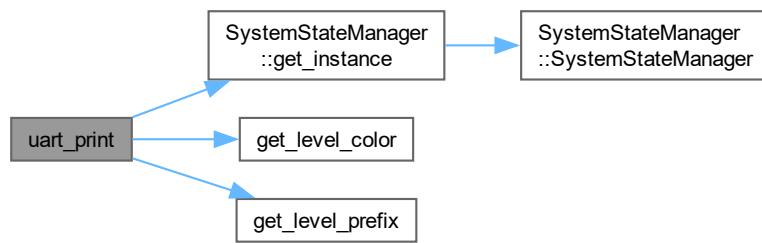
Parameters

<i>msg</i>	The message to print.
<i>uart</i>	The UART instance to use for printing.

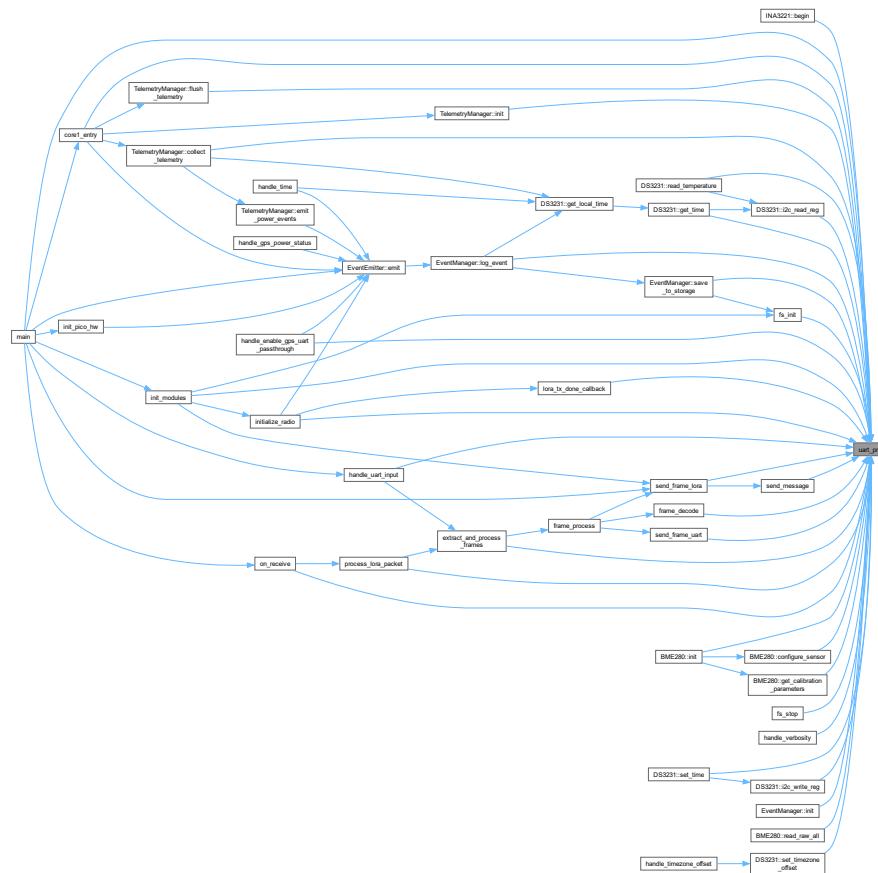
Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 60 of file [utils.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.89 utils.cpp

[Go to the documentation of this file.](#)

```

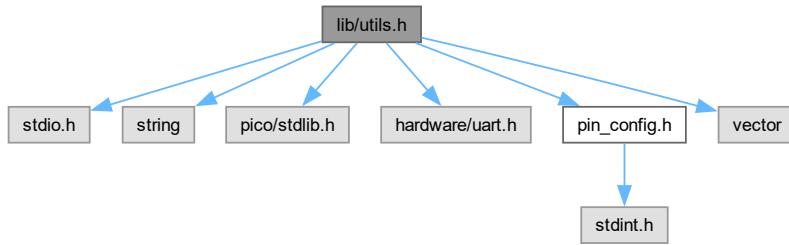
00001 #include "utils.h"
00002 #include "pico/multicore.h"
00003 #include "pico/sync.h"
00004 #include <vector>
00005 #include <queue>
00006 #include <string>
00007 #include <array>
00008 #include "system_state_manager.h"
00009
00014
00015
00017 namespace {
00018     mutex_t uart_mutex;
00019 }
00020
00021
00027 std::string get_level_color(VerbosityLevel level) {
00028     switch (level) {
00029         case VerbosityLevel::ERROR:    return ANSI_RED;
00030         case VerbosityLevel::WARNING: return ANSI_YELLOW;
00031         case VerbosityLevel::INFO:    return ANSI_GREEN;
00032         case VerbosityLevel::DEBUG:   return ANSI_BLUE;
00033         default:                      return "";
00034     }
00035 }
00036
00037
00043 std::string get_level_prefix(VerbosityLevel level) {
00044     switch (level) {
00045         case VerbosityLevel::ERROR:    return "ERROR: ";
00046         case VerbosityLevel::WARNING: return "WARNING: ";
00047         case VerbosityLevel::INFO:    return "INFO: ";
00048         case VerbosityLevel::DEBUG:   return "DEBUG: ";
00049         default:                      return "";
00050     }
00051 }
00052
00060 void uart_print(const std::string& msg, VerbosityLevel level, uart_inst_t* uart) {
00061     if (static_cast<int>(level) >
00062         static_cast<int>(SystemStateManager::get_instance().get_uart_verbosity())) {
00063         return;
00064     }
00065     static bool mutex_initiated = false;
00066     if (!mutex_initiated) {
00067         mutex_init(&uart_mutex);
00068         mutex_initiated = true;
00069     }
00070     uint32_t timestamp = to_ms_since_boot(get_absolute_time());
00071     uint core_num = get_core_num();
00073
00074     std::string color = get_level_color(level);
00075     std::string prefix = get_level_prefix(level);
00076     std::string msg_to_send = "[" + std::to_string(timestamp) + "ms] - Core " +
00077                             std::to_string(core_num) + ":" + color + prefix + ANSI_RESET + msg + "\r\n";
00078
00079     mutex_enter_blocking(&uart_mutex);
00080     uart_puts(uart, msg_to_send.c_str());
00081     mutex_exit(&uart_mutex);
00082
00083 }
```

9.90 lib/utils.h File Reference

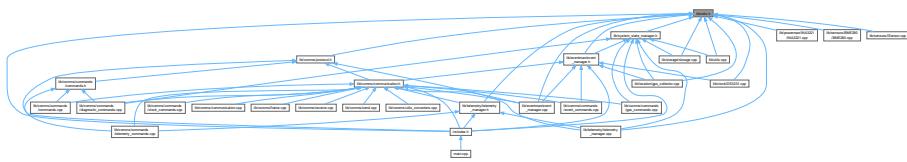
Utility functions and definitions for the Kubisat firmware.

```
#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` : `uint8_t` {

 `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`, `uart_inst_t` *`uart`=`DEBUG_UART_PORT`)
Prints a message to UART with timestamp and formatting.

9.90.1 Detailed Description

Utility functions and definitions for the Kabisat firmware.

Contains UART logging, color definitions, and CRC calculations

Definition in file [utils.h](#).

9.90.2 Macro Definition Documentation

9.90.2.1 ANSI_RED

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

ANSI escape codes for terminal color output.

Definition at line [20](#) of file [utils.h](#).

9.90.2.2 ANSI_GREEN

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

Definition at line [21](#) of file [utils.h](#).

9.90.2.3 ANSI_YELLOW

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

Definition at line [22](#) of file [utils.h](#).

9.90.2.4 ANSI_BLUE

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

Definition at line [23](#) of file [utils.h](#).

9.90.2.5 ANSI_RESET

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

Definition at line [24](#) of file [utils.h](#).

9.90.3 Enumeration Type Documentation

9.90.3.1 VerbosityLevel

```
enum class VerbosityLevel : uint8_t [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](#).

9.90.4 Function Documentation

9.90.4.1 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

<i>msg</i>	The message to print
<i>level</i>	Message verbosity level
<i>uart</i>	The UART port to use

Prints a message to UART with timestamp and formatting.

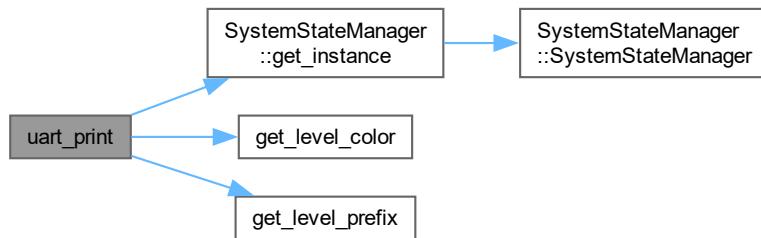
Parameters

<i>msg</i>	The message to print.
<i>uart</i>	The UART instance to use for printing.

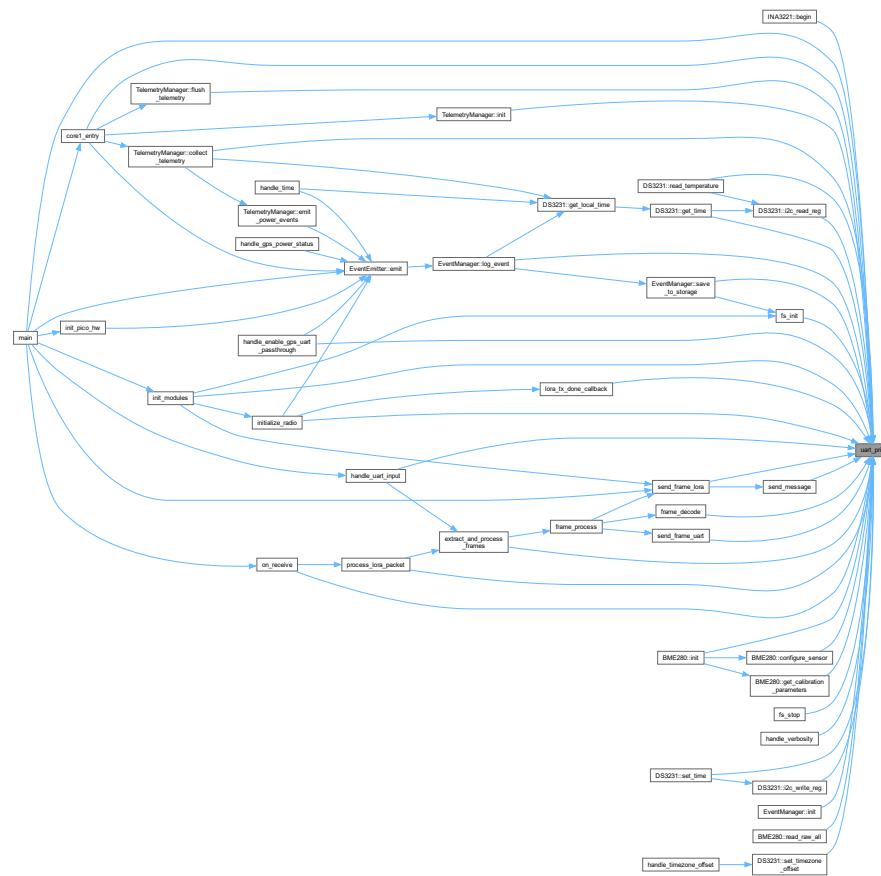
Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line [60](#) of file [utils.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



9.91 utils.h

[Go to the documentation of this file.](#)

```

00001 #ifndef UTILS_H
00002 #define UTILS_H
00003
00004 #include <stdio.h>
00005 #include <string>
00006 #include "pico/stl.h"
00007 #include "hardware/uart.h"
00008 #include "pin_config.h"
00009 #include <vector>
00010
00011
00017
00018
00020 #define ANSI_RED      "\033[31m"
00021 #define ANSI_GREEN    "\033[32m"
00022 #define ANSI_YELLOW   "\033[33m"
00023 #define ANSI_BLUE     "\033[34m"
00024 #define ANSI_RESET    "\033[0m"
00025
00026
00030 enum class VerbosityLevel : uint8_t {
00031     SILENT = 0,
00032     ERROR = 1,
00033     WARNING = 2,
00034     INFO = 3,
00035     DEBUG = 4
00036 };
00037
00038

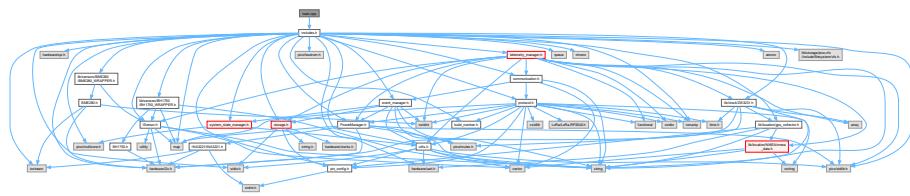
```

```
00045 void uart_print(const std::string& msg,
00046                 VerbosityLevel level,
00047                 uart_inst_t* uart = DEBUG_UART_PORT);
00048
00049
00050 #endif
```

9.92 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 ()`
- `SystemOperatingMode define_system_operating_mode ()`
- `int main ()`

9.92.1 Macro Definition Documentation

9.92.1.1 LOG_FILENAME

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

Definition at line 3 of file [main.cpp](#).

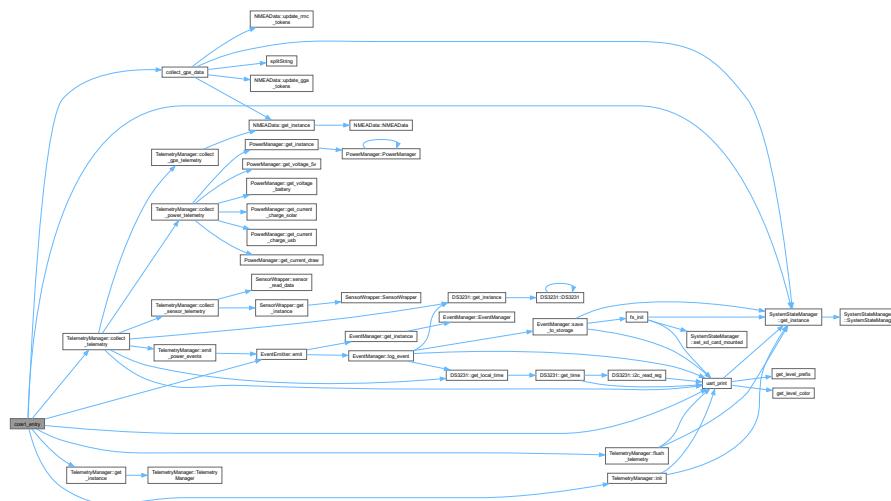
9.92.2 Function Documentation

9.92.2.1 core1_entry()

```
void core1_entry ()
```

Definition at line 5 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

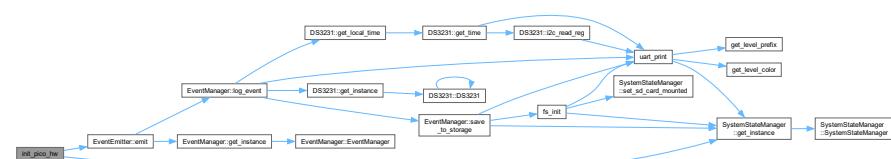


9.92.2.2 init_pico_hw()

```
bool init_pico_hw ()
```

Definition at line 40 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

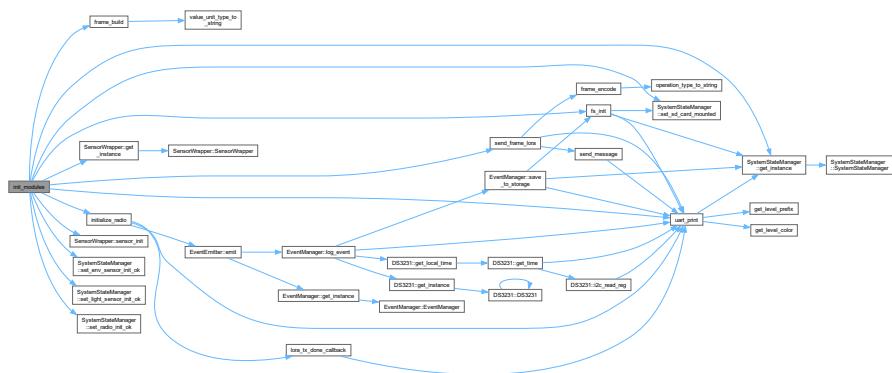


9.92.2.3 init_modules()

```
bool init_modules ()
```

Definition at line 86 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

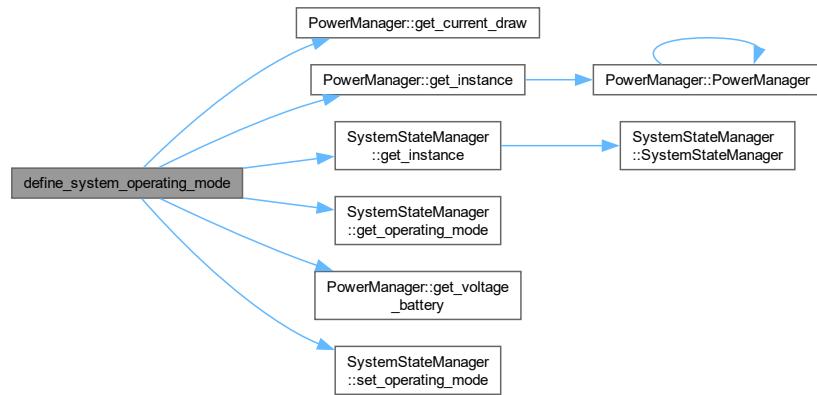


9.92.2.4 define_system_operating_mode()

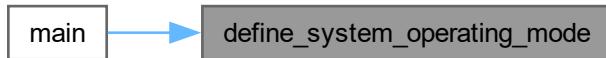
```
SystemOperatingMode define_system_operating_mode ()
```

Definition at line 147 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

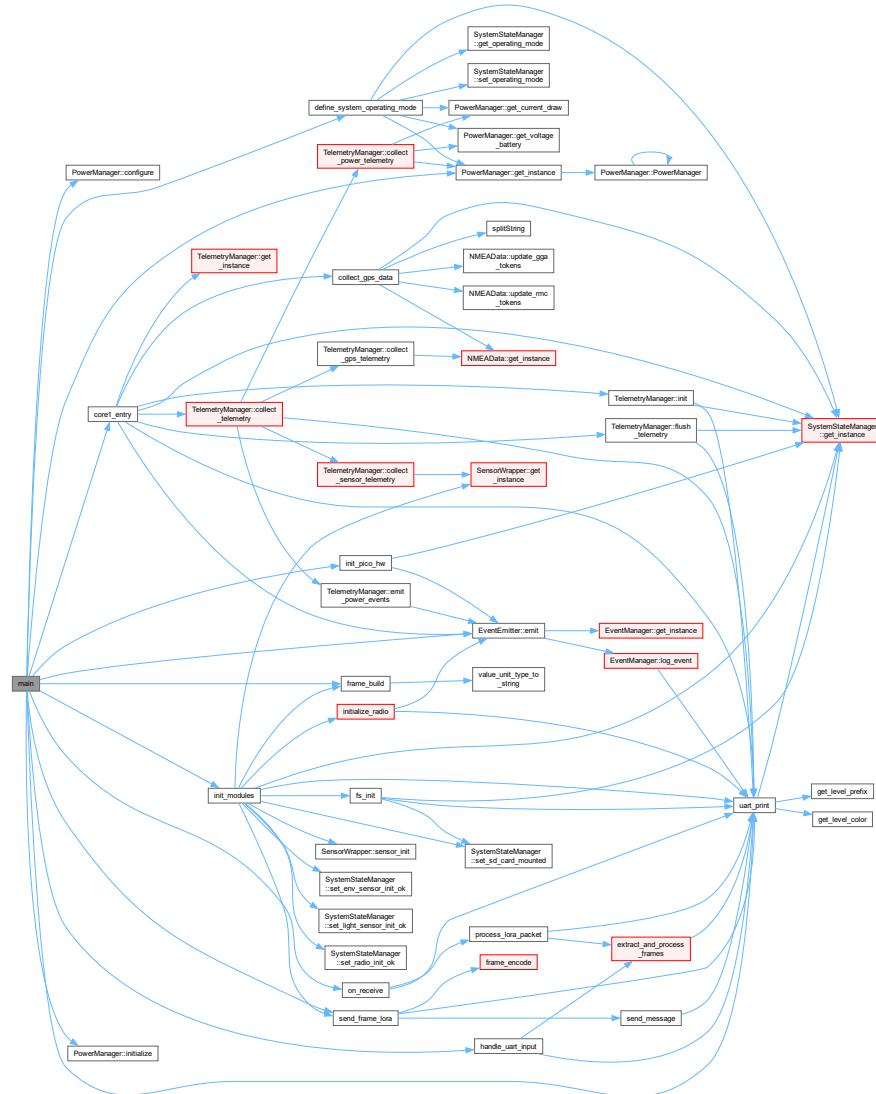


9.92.2.5 main()

```
int main ()
```

Definition at line 164 of file [main.cpp](#).

Here is the call graph for this function:



9.93 main.cpp

[Go to the documentation of this file.](#)

```

00001 #include "includes.h"
00002
00003 #define LOG_FILENAME "/log.txt"
00004
00005 void core1_entry() {
00006     uart_print("Starting core 1", VerbosityLevel::DEBUG);
00007     EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::CORE1_START);
00008
00009     uint32_t last telemetry_time = 0;
00010     uint32_t telemetry_collection_counter = 0;
00011
00012     TelemetryManager::get_instance().init();
00013
00014     while (true) {
00015         collect_gps_data();
00016
00017         uint32_t currentTime = to_ms_since_boot(get_absolute_time());
00018

```

```

00019     if (TelemetryManager::get_instance().is_telemetry_collection_time(currentTime,
00020         last_telemetry_time)) {
00021         TelemetryManager::get_instance().collect_telemetry();
00022         telemetry_collection_counter++;
00023
00024         if
00025             (TelemetryManager::get_instance().is_telemetry_flush_time(telemetry_collection_counter)) {
00026                 TelemetryManager::get_instance().flush_telemetry();
00027                 telemetry_collection_counter = 0;
00028                 uart_print("Telemetry flushed to SD", VerbosityLevel::INFO);
00029             }
00030
00031         if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
00032             sleep_ms(100);
00033             uart_print("Entering BOOTSEL mode...", VerbosityLevel::WARNING);
00034             reset_usb_boot(0, 0);
00035         }
00036
00037         sleep_ms(10);
00038     }
00039
00040 bool init_pico_hw() {
00041     stdio_init_all();
00042
00043     uart_init(DEBUG_UART_PORT, DEBUG_UART_BAUD_RATE);
00044     gpio_set_function(DEBUG_UART_TX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_TX_PIN));
00045     gpio_set_function(DEBUG_UART_RX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_RX_PIN));
00046
00047     uart_init(GPS_UART_PORT, GPS_UART_BAUD_RATE);
00048     gpio_set_function(GPS_UART_TX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_TX_PIN));
00049     gpio_set_function(GPS_UART_RX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_RX_PIN));
00050
00051     gpio_init(PICO_DEFAULT_LED_PIN);
00052     gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
00053
00054     gpio_init(PICO_DEFAULT_LED_PIN);
00055     gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
00056     gpio_put(PICO_DEFAULT_LED_PIN, true);
00057
00058     i2c_init(MAIN_I2C_PORT, 400 * 1000);
00059     gpio_set_function(MAIN_I2C_SCL_PIN, GPIO_FUNC_I2C);
00060     gpio_set_function(MAIN_I2C_SDA_PIN, GPIO_FUNC_I2C);
00061     gpio_pull_up(MAIN_I2C_SCL_PIN);
00062     gpio_pull_up(MAIN_I2C_SDA_PIN);
00063
00064     gpio_init(GPS_POWER_ENABLE_PIN);
00065     gpio_set_dir(GPS_POWER_ENABLE_PIN, GPIO_OUT);
00066     gpio_put(GPS_POWER_ENABLE_PIN, true);
00067
00068     i2c_init(SENSORS_I2C_PORT, 400 * 1000);
00069     gpio_set_function(SENSORS_I2C_SCL_PIN, GPIO_FUNC_I2C);
00070     gpio_set_function(SENSORS_I2C_SDA_PIN, GPIO_FUNC_I2C);
00071     gpio_pull_up(SENSORS_I2C_SCL_PIN);
00072     gpio_pull_up(SENSORS_I2C_SDA_PIN);
00073     gpio_init(SENSORS_POWER_ENABLE_PIN);
00074     gpio_set_dir(SENSORS_POWER_ENABLE_PIN, GPIO_OUT);
00075     gpio_put(SENSORS_POWER_ENABLE_PIN, true);
00076
00077     SystemStateManager::get_instance();
00078
00079     EventEmitter::emit(EventGroup::GPS, GPSEvent::POWER_ON);
00080
00081     system("color");
00082
00083     return true;
00084 }
00085
00086 bool init_modules() {
00087     bool radio_init_status = initialize_radio();
00088     SystemStateManager::get_instance().set_radio_init_ok(radio_init_status);
00089
00090     bool sd_init_status = fs_init();
00091     SystemStateManager::get_instance().set_sd_card_mounted(sd_init_status);
00092
00093     if (sd_init_status) {
00094         FILE *fp = fopen(LOG_FILENAME, "w");
00095         if (fp) {
00096             uart_print("Log file opened.", VerbosityLevel::DEBUG);
00097             int bytes_written = fprintf(fp, "System init started.\n");
00098             uart_print("Written " + std::to_string(bytes_written) + " bytes.", VerbosityLevel::DEBUG);
00099             int close_status = fclose(fp);
00100             uart_print("Close file status: " + std::to_string(close_status), VerbosityLevel::DEBUG);
00101
00102             struct stat file_stat;
00103             if (stat(LOG_FILENAME, &file_stat) == 0) {

```

```

00104         size_t file_size = file_stat.st_size;
00105         uart_print("File size: " + std::to_string(file_size) + " bytes",
00106         VerbosityLevel::DEBUG);
00107     } else {
00108         uart_print("Failed to get file size", VerbosityLevel::ERROR);
00109     }
00110     uart_print("File path: /" + std::string(LOG_FILENAME), VerbosityLevel::DEBUG);
00111 } else {
00112     uart_print("Failed to open log file for writing.", VerbosityLevel::ERROR);
00113 }
00114 }
00115
00116 if (sd_init_status) {
00117     uart_print("SD card init: OK", VerbosityLevel::DEBUG);
00118 } else {
00119     uart_print("SD card init: FAILED", VerbosityLevel::ERROR);
00120 }
00121
00122 if (radio_init_status) {
00123     uart_print("Radio init: OK", VerbosityLevel::DEBUG);
00124 } else {
00125     uart_print("Radio init: FAILED", VerbosityLevel::ERROR);
00126 }
00127
00128 Frame boot = frame_build(OperationType::RES, 0, 0, "HELLO");
00129 send_frame_lora.boot);
00130
00131 uart_print("Initializing sensors...", VerbosityLevel::DEBUG);
00132
00133 SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00134 bool light_sensor_init = sensor_wrapper.sensor_init(SensorType::LIGHT, SENSORS_I2C_PORT);
00135 SystemStateManager::get_instance().set_light_sensor_init_ok(light_sensor_init);
00136
00137 bool env_sensor_init = sensor_wrapper.sensor_init(SensorType::ENVIRONMENT, SENSORS_I2C_PORT);
00138 SystemStateManager::get_instance().set_env_sensor_init_ok(env_sensor_init);
00139
00140 if (!light_sensor_init || !env_sensor_init) {
00141     uart_print("One or more sensors failed to initialize", VerbosityLevel::WARNING);
00142 }
00143
00144 return sd_init_status && radio_init_status;
00145 }
00146
00147 SystemOperatingMode define_system_operating_mode() {
00148     // If system is running but measured battery voltage is below this threshold it means that power
00149     // is sourced from USB
00150     static constexpr float BAT_VOLTAGE_THRESHOLD = 2.4f;
00151     // If system is running but measured current discharge is below this threshold it means that power
00152     // is sourced from USB
00153     static constexpr uint8_t current_discharge_threshold = 40;
00154     float battery_voltage = PowerManager::get_instance().get_voltage_battery();
00155     uint8_t current_discharge = PowerManager::get_instance().get_current_draw();
00156
00157     if (battery_voltage < BAT_VOLTAGE_THRESHOLD && current_discharge < current_discharge_threshold) {
00158         SystemStateManager::get_instance().set_operating_mode(SystemOperatingMode::USB_POWERED);
00159     } else {
00160         SystemStateManager::get_instance().set_operating_mode(SystemOperatingMode::BATTERY_POWERED);
00161     }
00162 }
00163
00164 int main()
00165 {
00166     init_pico_hw();
00167     sleep_ms(100);
00168     init_modules();
00169     EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::BOOT);
00170     sleep_ms(100);
00171
00172     bool power_manager_init_status = PowerManager::get_instance().initialize();
00173     if (power_manager_init_status) {
00174         PowerManager::get_instance().configure(CONTINUOUS_MODE, INA3221_REG_CONF_AVG_16);
00175     } else {
00176         uart_print("Power manager init error", VerbosityLevel::ERROR);
00177     }
00178
00179     SystemOperatingMode current_mode = define_system_operating_mode();
00180
00181     multicore_launch_core1(core1_entry);
00182
00183     gpio_put(PICO_DEFAULT_LED_PIN, false);
00184
00185     std::string mode_string = (current_mode == SystemOperatingMode::USB_POWERED) ? "USB" : "BATTERY";
00186     uart_print("Operating mode: " + mode_string, VerbosityLevel::WARNING);
00187     Frame boot = frame_build(OperationType::RES, 0, 0, "START_MODE_" + mode_string);

```

```
00188     send_frame_lora(boot);
00189
00190     std::string boot_string = "System init completed @ " +
00191         std::to_string(to_ms_since_boot(get_absolute_time())) + " ms";
00192     uart_print(boot_string, VerbosityLevel::WARNING);
00193
00194     gpio_put(PICO_DEFAULT_LED_PIN, true);
00195
00196     while (true)
00197     {
00198         int packet_size = LoRa.parse_packet();
00199         if (packet_size)
00200         {
00201             on_receive(packet_size);
00202         }
00203
00204         handle_uart_input();
00205
00206     return 0;
00207 }
```

Index

_BH1750_DEFAULT_MTREG
 Constants, [78](#)
_BH1750_DEVICE_ID
 Constants, [78](#)
_BH1750_MTREG_MAX
 Constants, [78](#)
_BH1750_MTREG_MIN
 Constants, [78](#)
__attribute__
 Event Management, [56](#), [58](#)
_filterRes
 INA3221, [142](#)
_i2c
 INA3221, [142](#)
_i2c_addr
 BH1750, [98](#)
 INA3221, [142](#)
_masken_reg
 INA3221, [142](#)
_read
 INA3221, [141](#)
_shuntRes
 INA3221, [142](#)
_write
 INA3221, [141](#)
~ISensor
 ISensor, [144](#)
~TelemetryManager
 TelemetryManager, [177](#)

ADDR_SDO_HIGH
 BME280, [105](#)
ADDR_SDO_LOW
 BME280, [105](#)
altitude
 TelemetryRecord, [183](#)
ANSI_BLUE
 utils.h, [308](#)
ANSI_GREEN
 utils.h, [308](#)
ANSI_RED
 utils.h, [308](#)
ANSI_RESET
 utils.h, [308](#)
ANSI_YELLOW
 utils.h, [308](#)
avg_mode
 INA3221::conf_reg_t, [124](#)

BATTERY_FULL

 Event Management, [54](#)
 PowerManager, [157](#)
BATTERY_LOW
 Event Management, [54](#)
 PowerManager, [157](#)
BATTERY_NORMAL
 Event Management, [54](#)
BATTERY_POWERED
 System State Manager, [85](#)
battery_voltage
 TelemetryRecord, [181](#)
begin
 BH1750 Light Sensor, [75](#)
 Configuration Functions, [62](#)
BH1750, [97](#)
 _i2c_addr, [98](#)
 BH1750 Light Sensor, [75](#)
 i2c_port_, [98](#)
 BH1750 Light Sensor, [74](#)
 begin, [75](#)
 BH1750, [75](#)
 configure, [75](#)
 get_light_level, [76](#)
 write8, [76](#)
 BH1750Wrapper, [99](#)
 BH1750Wrapper, [100](#)
 configure, [101](#)
 get_address, [102](#)
 get_i2c_addr, [100](#)
 get_type, [101](#)
 init, [100](#)
 initialized_, [102](#)
 is_initialized, [101](#)
 read_data, [100](#)
 sensor_, [102](#)
 BME280, [103](#)
 ADDR_SDO_HIGH, [105](#)
 ADDR_SDO_LOW, [105](#)
 BME280, [107](#)
 calib_params, [113](#)
 configure_sensor, [112](#)
 convert_humidity, [110](#)
 convert_pressure, [109](#)
 convert_temperature, [109](#)
 device_addr, [113](#)
 get_calibration_parameters, [112](#)
 HUMIDITY_OVERSAMPLING, [107](#)

i2c_port, 113
 init, 108
 initialized_, 114
 NORMAL_MODE, 107
 NUM_CALIB_PARAMS, 107
 NUM_HUM_CALIB_PARAMS, 107
 OSR_X0, 105
 OSR_X1, 105
 OSR_X16, 105
 OSR_X2, 105
 OSR_X4, 105
 OSR_X8, 105
 Oversampling, 105
 PRESSURE_OVERSAMPLING, 107
 read_raw_all, 108
 read_register, 110, 111
 REG_CONFIG, 106
 REG_CTRL_HUM, 106
 REG_CTRL_MEAS, 106
 REG_DIG_H1, 106
 REG_DIG_H2, 106
 REG_DIG_H3, 106
 REG_DIG_H4, 106
 REG_DIG_H5, 106
 REG_DIG_H6, 106
 REG_DIG_P1_LSB, 106
 REG_DIG_P1_MSB, 106
 REG_DIG_P2_LSB, 106
 REG_DIG_P2_MSB, 106
 REG_DIG_P3_LSB, 106
 REG_DIG_P3_MSB, 106
 REG_DIG_P4_LSB, 106
 REG_DIG_P4_MSB, 106
 REG_DIG_P5_LSB, 106
 REG_DIG_P5_MSB, 106
 REG_DIG_P6_LSB, 106
 REG_DIG_P6_MSB, 106
 REG_DIG_P7_LSB, 106
 REG_DIG_P7_MSB, 106
 REG_DIG_P8_LSB, 106
 REG_DIG_P8_MSB, 106
 REG_DIG_P9_LSB, 106
 REG_DIG_P9_MSB, 106
 REG_DIG_T1_LSB, 106
 REG_DIG_T1_MSB, 106
 REG_DIG_T2_LSB, 106
 REG_DIG_T2_MSB, 106
 REG_DIG_T3_LSB, 106
 REG_DIG_T3_MSB, 106
 REG_HUMIDITY_MSB, 106
 REG_PRESSURE_MSB, 106
 REG_RESET, 106
 REG_TEMPERATURE_MSB, 106
 reset, 108
 t_fine, 114
 TEMPERATURE_OVERSAMPLING, 107
 write_register, 110
 BME280CalibParam, 114
 dig_h1, 117
 dig_h2, 118
 dig_h3, 118
 dig_h4, 118
 dig_h5, 118
 dig_h6, 118
 dig_p1, 116
 dig_p2, 116
 dig_p3, 116
 dig_p4, 116
 dig_p5, 117
 dig_p6, 117
 dig_p7, 117
 dig_p8, 117
 dig_p9, 117
 dig_t1, 116
 dig_t2, 116
 dig_t3, 116
 BME280Wrapper, 119
 BME280Wrapper, 120
 configure, 121
 get_address, 122
 get_type, 121
 init, 120
 initialized_, 122
 is_initialized, 121
 read_data, 120
 sensor_, 122
 BOOL
 Protocol, 38
 BOOT
 Event Management, 54
 BUFFER_SIZE
 pin_config.h, 252
 BUILD_NUMBER
 build_number.h, 185
 build_number.h, 185
 BUILD_NUMBER, 185
 build_version
 TelemetryRecord, 181
 build_version_command_id
 Diagnostic Commands, 24
 bus_conv_time
 INA3221::conf_reg_t, 123
 calib_params
 BME280, 113
 CELSIUS
 Protocol, 38
 century
 ds3231_data_t, 130
 ch1_en
 INA3221::conf_reg_t, 124
 ch2_en
 INA3221::conf_reg_t, 124
 ch3_en
 INA3221::conf_reg_t, 124
 CHANGED
 Event Management, 56

charge_current_solar
TelemetryRecord, 181

charge_current_usb
TelemetryRecord, 181

CHARGING
Event Management, 54

CLOCK
Event Management, 54

Clock Commands, 3

Clock Management Commands, 13
handle_get_internal_temperature, 15
handle_time, 13
handle_timezone_offset, 14

clock_commands.cpp
clock_commands_group_id, 195
internal_temperature_command_id, 196
time_command_id, 195
timezone_offset_command_id, 196

clock_commands_group_id
clock_commands.cpp, 195

clock_mutex
DS3231, 127

ClockEvent
Event Management, 55

CMD
Command System, 17

collect_gps_data
Location, 60

collect_gps_telemetry
Telemetry Manager, 89

collect_power_telemetry
Telemetry Manager, 87

collect_sensor_telemetry
Telemetry Manager, 89

collect_telemetry
Telemetry Manager, 91

command
Frame, 139

Command System, 16
CMD, 17
command_handlers, 18
CommandHandler, 17
CommandMap, 17
execute_command, 17

command_handlers
Command System, 18

CommandHandler
Command System, 17
frame.cpp, 222

CommandMap
Command System, 17

commands_list_command_id
Diagnostic Commands, 24

COMMS
Event Management, 54

CommsEvent
Event Management, 54

communication.cpp
initialize_radio, 213
lora_tx_done_callback, 213

communication.h
handle_uart_input, 218
initialize_radio, 216
lora_tx_done_callback, 216
on_receive, 217
send_frame_lora, 220
send_frame_uart, 219
send_message, 218

Configuration Functions, 61
begin, 62
get_die_id, 63
get_manufacturer_id, 62
INA3221, 62
read_register, 64
set_averaging_mode, 65
set_mode_continuous, 64
set_mode_triggered, 65

configure
BH1750 Light Sensor, 75
BH1750Wrapper, 101
BME280Wrapper, 121
ISensor, 145
Power Management, 73

configure_sensor
BME280, 112

Constants, 77
_BH1750_DEFAULT_MTREG, 78
_BH1750_DEVICE_ID, 78
_BH1750_MTREG_MAX, 78
_BH1750_MTREG_MIN, 78

CONTINUOUS_HIGH_RES_MODE
Types, 79

CONTINUOUS_HIGH_RES_MODE_2
Types, 79

CONTINUOUS_LOW_RES_MODE
Types, 79

CONTINUOUS_MODE
INA3221.h, 263

conv_ready
INA3221::masken_reg_t, 146

convert_humidity
BME280, 110

convert_pressure
BME280, 109

convert_temperature
BME280, 109

core1_entry
main.cpp, 312

CORE1_START
Event Management, 54

CORE1_STOP
Event Management, 54

course
TelemetryRecord, 183

credits, 1

credits.md, 186

crit_alert_ch1
 INA3221::masken_reg_t, 147
 crit_alert_ch2
 INA3221::masken_reg_t, 147
 crit_alert_ch3
 INA3221::masken_reg_t, 147
 crit_alert_latch_en
 INA3221::masken_reg_t, 147

 DATA_READY
 Event Management, 55
 date
 ds3231_data_t, 129
 TelemetryRecord, 183
 DATETIME
 Protocol, 38
 day
 ds3231_data_t, 129
 days_of_week
 DS3231.h, 193
 DEBUG
 utils.h, 308
 DEBUG_UART_BAUD_RATE
 pin_config.h, 250
 DEBUG_UART_PORT
 pin_config.h, 250
 DEBUG_UART_RX_PIN
 pin_config.h, 250
 DEBUG_UART_TX_PIN
 pin_config.h, 250
 DEFAULT_FLUSH_THRESHOLD
 Telemetry Manager, 87
 TelemetryManager, 178
 DEFAULT_SAMPLE_INTERVAL_MS
 Telemetry Manager, 86
 TelemetryManager, 178
 define_system_operating_mode
 main.cpp, 313
 DELIMITER
 protocol.h, 225
 device_addr
 BME280, 113
 Diagnostic Commands, 19
 build_version_command_id, 24
 commands_list_command_id, 24
 diagnostic_commands_group_id, 24
 enter_bootloader_command_id, 25
 handle_enter_bootloader_mode, 23
 handle_get_build_version, 20
 handle_get_commands_list, 19
 handle_get_power_mode, 21
 handle_get_uptime, 20
 handle_verbosity, 22
 power_mode_command_id, 24
 uptime_command_id, 25
 verbosity_command_id, 25
 diagnostic_commands_group_id
 Diagnostic Commands, 24
 dig_h1
 BME280CalibParam, 117
 dig_h2
 BME280CalibParam, 118
 dig_h3
 BME280CalibParam, 118
 dig_h4
 BME280CalibParam, 118
 dig_h5
 BME280CalibParam, 118
 dig_h6
 BME280CalibParam, 118
 dig_p1
 BME280CalibParam, 116
 dig_p2
 BME280CalibParam, 116
 dig_p3
 BME280CalibParam, 116
 dig_p4
 BME280CalibParam, 116
 dig_p5
 BME280CalibParam, 117
 dig_p6
 BME280CalibParam, 117
 dig_p7
 BME280CalibParam, 117
 dig_p8
 BME280CalibParam, 117
 dig_p9
 BME280CalibParam, 117
 dig_t1
 BME280CalibParam, 116
 dig_t2
 BME280CalibParam, 116
 dig_t3
 BME280CalibParam, 116
 direction
 Frame, 138
 discharge_current
 TelemetryRecord, 182
 DISCHARGING
 Event Management, 54
 DS3231, 124
 clock_mutex_, 127
 DS3231, 126
 ds3231_addr, 127
 i2c, 127
 last_sync_time_, 128
 operator=, 127
 RTC clock, 43
 sync_interval_minutes_, 128
 timezone_offset_minutes_, 128
 DS3231.h
 days_of_week, 193
 DS3231_CONTROL_REG, 192
 DS3231_CONTROL_STATUS_REG, 193
 DS3231_DATE_REG, 192
 DS3231_DAY_REG, 192
 DS3231_DEVICE_ADRESS, 191

DS3231_HOURS_REG, 192
DS3231_MINUTES_REG, 191
DS3231_MONTH_REG, 192
DS3231_SECONDS_REG, 191
DS3231_TEMPERATURE_LSB_REG, 193
DS3231_TEMPERATURE_MSB_REG, 193
DS3231_YEAR_REG, 192
FRIDAY, 193
MONDAY, 193
SATURDAY, 193
SUNDAY, 193
THURSDAY, 193
TUESDAY, 193
WEDNESDAY, 193
ds3231_addr
 DS3231, 127
DS3231_CONTROL_REG
 DS3231.h, 192
DS3231_CONTROL_STATUS_REG
 DS3231.h, 193
ds3231_data_t, 128
 century, 130
 date, 129
 day, 129
 hours, 129
 minutes, 129
 month, 129
 seconds, 129
 year, 130
DS3231_DATE_REG
 DS3231.h, 192
DS3231_DAY_REG
 DS3231.h, 192
DS3231_DEVICE_ADDRESS
 DS3231.h, 191
DS3231_HOURS_REG
 DS3231.h, 192
DS3231_MINUTES_REG
 DS3231.h, 191
DS3231_MONTH_REG
 DS3231.h, 192
DS3231_SECONDS_REG
 DS3231.h, 191
DS3231_TEMPERATURE_LSB_REG
 DS3231.h, 193
DS3231_TEMPERATURE_MSB_REG
 DS3231.h, 193
DS3231_YEAR_REG
 DS3231.h, 192
emit
 EventEmitter, 130
emit_power_events
 Telemetry Manager, 88
enter_bootloader_command_id
 Diagnostic Commands, 25
env_sensor_init_status
 SystemStateManager, 174
ENVIRONMENT
 Sensors, 80
ERR
 Protocol, 38
ERROR
 Event Management, 55
 utils.h, 308
error_code_to_string
 Utility Converters, 41
ErrorCode
 Protocol, 37
event
 event_manager.h, 241
 EventLog, 132
Event Commands, 25
 event_commands_group_id, 27
 event_count_command_id, 27
 handle_get_event_count, 26
 handle_get_last_events, 25
 last_events_command_id, 27
Event Management, 51
 __attribute__, 56, 58
 BATTERY_FULL, 54
 BATTERY_LOW, 54
 BATTERY_NORMAL, 54
 BOOT, 54
 CHANGED, 56
 CHARGING, 54
 CLOCK, 54
 ClockEvent, 55
 COMMS, 54
 CommsEvent, 54
 CORE1_START, 54
 CORE1_STOP, 54
 DATA_READY, 55
 DISCHARGING, 54
 ERROR, 55
 EVENT_BUFFER_SIZE, 53
 EVENT_FLUSH_THRESHOLD, 53
 EVENT_LOG_FILE, 53
 EventGroup, 53
 get_event, 57
 GPS, 54
 GPS_SYNC, 56
 GPS_SYNC_DATA_NOT_READY, 56
 GPSEvent, 55
 init, 56
 LOCK, 55
 log_event, 56
 LOST, 55
 MSG_RECEIVED, 55
 MSG_SENT, 55
 PASS_THROUGH_END, 55
 PASS_THROUGH_START, 55
 POWER, 54
 POWER_FALLING, 54
 POWER_OFF, 55
 POWER_ON, 55
 PowerEvent, 54

RADIO_ERROR, 55
 RADIO_INIT, 55
 save_to_storage, 57
 SHUTDOWN, 54
 SOLAR_ACTIVE, 54
 SOLAR_INACTIVE, 54
 SYSTEM, 54
 SystemEvent, 54
 UART_ERROR, 55
 USB_CONNECTED, 54
 USB_DISCONNECTED, 54
 WATCHDOG_RESET, 54
EVENT_BUFFER_SIZE
 Event Management, 53
event_commands_group_id
 Event Commands, 27
event_count_command_id
 Event Commands, 27
EVENT_FLUSH_THRESHOLD
 Event Management, 53
EVENT_LOG_FILE
 Event Management, 53
event_manager.h
 event, 241
 group, 241
 id, 241
 timestamp, 241
eventCount
 EventManager, 136
EventEmitter, 130
 emit, 130
EventGroup
 Event Management, 53
EventLog, 131
 event, 132
 group, 132
 id, 132
 timestamp, 132
EventManager, 133
 eventCount, 136
 EventManager, 134
 eventMutex, 137
 events, 136
 eventsSinceFlush, 137
 get_event_count, 136
 get_instance, 135
 nextEventId, 137
 operator=, 135
 writeIndex, 137
eventMutex
 EventManager, 137
events
 EventManager, 136
eventsSinceFlush
 EventManager, 137
execute_command
 Command System, 17
extract_and_process_frames
 receive.cpp, 228
FAIL_TO_SET
 Protocol, 37
fix_quality
 TelemetryRecord, 183
flush_sensor_data
 TelemetryManager, 177
flush_telemetry
 Telemetry Manager, 92
flush_threshold
 TelemetryManager, 179
footer
 Frame, 139
Frame, 137
 command, 139
 direction, 138
 footer, 139
 group, 138
 header, 138
 operationType, 138
 unit, 139
 value, 139
Frame Handling, 32
 frame_build, 35
 frame_decode, 33
 frame_encode, 32
 frame_process, 34
frame.cpp
 CommandHandler, 222
FRAME_BEGIN
 protocol.h, 225
frame_build
 Frame Handling, 35
frame_decode
 Frame Handling, 33
frame_encode
 Frame Handling, 32
FRAME_END
 protocol.h, 225
frame_process
 Frame Handling, 34
FRIDAY
 DS3231.h, 193
fs_init
 Storage, 83
fs_stop
 Storage, 84
GET
 Protocol, 38
get_address
 BH1750Wrapper, 102
 BME280Wrapper, 122
 ISensor, 145
get_available_sensors
 SensorWrapper, 161
get_calibration_parameters
 BME280, 112

get_current_charge_solar
 Power Management, 72

get_current_charge_total
 Power Management, 73

get_current_charge_usb
 Power Management, 72

get_current_draw
 Power Management, 72

get_current_ma
 Measurement Functions, 67

get_die_id
 Configuration Functions, 63

get_event
 Event Management, 57

get_event_count
 EventManager, 136

get_gga_tokens
 NMEAData, 152

get_i2c_addr
 BH1750Wrapper, 100

get_instance
 EventManager, 135
 NMEAData, 150
 Power Management, 69
 RTC clock, 43
 SensorWrapper, 161
 SystemStateManager, 165
 TelemetryManager, 177

get_last_sensor_record_csv
 Telemetry Manager, 94

get_last_telemetry_record_csv
 Telemetry Manager, 94

get_level_color
 utils.cpp, 302

get_level_prefix
 utils.cpp, 303

get_light_level
 BH1750 Light Sensor, 76

get_local_time
 RTC clock, 48

get_manufacturer_id
 Configuration Functions, 62

get_operating_mode
 SystemStateManager, 172

get_rmc_tokens
 NMEAData, 152

get_sensor
 Sensors, 82

get_shunt_voltage
 Measurement Functions, 66

get_telemetry_buffer_count
 TelemetryManager, 178

get_telemetry_buffer_write_index
 TelemetryManager, 178

get_time
 RTC clock, 44

get_timezone_offset
 RTC clock, 46

get_type
 BH1750Wrapper, 101
 BME280Wrapper, 121
 ISensor, 145

get_uart_verbosity
 SystemStateManager, 169

get_voltage
 Measurement Functions, 68

get_voltage_5v
 Power Management, 71

get_voltage_battery
 Power Management, 71

get_voltage_solar
 Power Management, 71

gga_mutex_
 NMEAData, 153

gga_tokens_
 NMEAData, 152

GPS
 Event Management, 54

GPS Commands, 28
 handle_enable_gps_uart_passthrough, 29
 handle_gps_power_status, 28

gps_collection_paused
 SystemStateManager, 173

gps_commands.cpp
 gps_commands_group_id, 208
 passthrough_command_id, 208
 power_status_command_id, 208

gps_commands_group_id
 gps_commands.cpp, 208

GPS_POWER_ENABLE_PIN
 pin_config.h, 251

GPS_SYNC
 Event Management, 56

GPS_SYNC_DATA_NOT_READY
 Event Management, 56

GPS_UART_BAUD_RATE
 pin_config.h, 251

GPS_UART_PORT
 pin_config.h, 251

GPS_UART_RX_PIN
 pin_config.h, 251

GPS_UART_TX_PIN
 pin_config.h, 251

GPSEvent
 Event Management, 55

group
 event_manager.h, 241
 EventLog, 132
 Frame, 138

handle_enable_gps_uart_passthrough
 GPS Commands, 29

handle_enter_bootloader_mode
 Diagnostic Commands, 23

handle_get_build_version
 Diagnostic Commands, 20

handle_get_commands_list

Diagnostic Commands, 19
handle_get_event_count
 Event Commands, 26
handle_get_internal_temperature
 Clock Management Commands, 15
handle_get_last_events
 Event Commands, 25
handle_get_last_sensor_record
 Telemetry Buffer Commands, 31
handle_get_last_telemetry_record
 Telemetry Buffer Commands, 30
handle_get_power_mode
 Diagnostic Commands, 21
handle_get_uptime
 Diagnostic Commands, 20
handle_gps_power_status
 GPS Commands, 28
handle_time
 Clock Management Commands, 13
handle_timezone_offset
 Clock Management Commands, 14
handle_uart_input
 communication.h, 218
 receive.cpp, 230
handle_verbosity
 Diagnostic Commands, 22
header
 Frame, 138
hours
 ds3231_data_t, 129
HUMIDITY
 Sensors, 81
humidity
 SensorDataRecord, 159
HUMIDITY_OVERSAMPLING
 BME280, 107
i2c
 DS3231, 127
i2c_port
 BME280, 113
i2c_port_
 BH1750, 98
i2c_read_reg
 RTC clock, 49
i2c_write_reg
 RTC clock, 50
id
 event_manager.h, 241
 EventLog, 132
INA3221, 139
 _filterRes, 142
 _i2c, 142
 _i2c_addr, 142
 _masken_reg, 142
 _read, 141
 _shuntRes, 142
 _write, 141
 Configuration Functions, 62
 INA3221 Power Monitor, 61
INA3221.h
 CONTINUOUS_MODE, 263
 INA3221_ADDR40_GND, 261
 INA3221_ADDR41_VCC, 261
 INA3221_ADDR42_SDA, 261
 INA3221_ADDR43_SCL, 261
 ina3221_addr_t, 261
 ina3221_avg_mode_t, 262
 INA3221_CH1, 261
 INA3221_CH2, 261
 INA3221_CH3, 261
 INA3221_CH_NUM, 263
 ina3221_ch_t, 261
 ina3221_op_mode_t, 262
 INA3221_REG_CH1_BUSV, 262
 INA3221_REG_CH1_CRIT_ALERT_LIM, 262
 INA3221_REG_CH1_SHUNTV, 262
 INA3221_REG_CH1_WARNING_ALERT_LIM, 262
 INA3221_REG_CH2_BUSV, 262
 INA3221_REG_CH2_CRIT_ALERT_LIM, 262
 INA3221_REG_CH2_SHUNTV, 262
 INA3221_REG_CH2_WARNING_ALERT_LIM, 262
 INA3221_REG_CH3_BUSV, 262
 INA3221_REG_CH3_CRIT_ALERT_LIM, 262
 INA3221_REG_CH3_SHUNTV, 262
 INA3221_REG_CH3_WARNING_ALERT_LIM, 262
 INA3221_REG_CONF, 262
 INA3221_REG_CONF_AVG_1, 262
 INA3221_REG_CONF_AVG_1024, 262
 INA3221_REG_CONF_AVG_128, 262
 INA3221_REG_CONF_AVG_16, 262
 INA3221_REG_CONF_AVG_256, 262
 INA3221_REG_CONF_AVG_4, 262
 INA3221_REG_CONF_AVG_512, 262
 INA3221_REG_CONF_AVG_64, 262
 INA3221_REG_DIE_ID, 262
 INA3221_REG_MANUF_ID, 262
 INA3221_REG_MASK_ENABLE, 262
 INA3221_REG_PWR_VALID_HI_LIM, 262
 INA3221_REG_PWR_VALID_LO_LIM, 262
 INA3221_REG_SHUNTV_SUM, 262
 INA3221_REG_SHUNTV_SUM_LIM, 262
 ina3221_reg_t, 261
 SHUNT_VOLTAGE_LSB_UV, 263
 TRIGGERED_MODE, 263
INA3221::conf_reg_t, 122
 avg_mode, 124
 bus_conv_time, 123
 ch1_en, 124
 ch2_en, 124
 ch3_en, 124
 mode_bus_en, 123
 mode_continious_en, 123
 mode_shunt_en, 123

reset, 124
shunt_conv_time, 123
INA3221::masken_reg_t, 146
conv_ready, 146
crit_alert_ch1, 147
crit_alert_ch2, 147
crit_alert_ch3, 147
crit_alert_latch_en, 147
pwr_valid_alert, 146
reserved, 148
shunt_sum_alert, 147
shunt_sum_en_ch1, 148
shunt_sum_en_ch2, 148
shunt_sum_en_ch3, 148
timing_ctrl_alert, 146
warn_alert_ch1, 147
warn_alert_ch2, 147
warn_alert_ch3, 146
warn_alert_latch_en, 147
ina3221_
 PowerManager, 157
INA3221_ADDR40_GND
 INA3221.h, 261
INA3221_ADDR41_VCC
 INA3221.h, 261
INA3221_ADDR42_SDA
 INA3221.h, 261
INA3221_ADDR43_SCL
 INA3221.h, 261
ina3221_addr_t
 INA3221.h, 261
ina3221_avg_mode_t
 INA3221.h, 262
INA3221_CH1
 INA3221.h, 261
INA3221_CH2
 INA3221.h, 261
INA3221_CH3
 INA3221.h, 261
INA3221_CH_NUM
 INA3221.h, 263
ina3221_ch_t
 INA3221.h, 261
ina3221_op_mode_t
 INA3221.h, 262
INA3221_REG_CH1_BUSV
 INA3221.h, 262
INA3221_REG_CH1_CRIT_ALERT_LIM
 INA3221.h, 262
INA3221_REG_CH1_SHUNTV
 INA3221.h, 262
INA3221_REG_CH1_WARNING_ALERT_LIM
 INA3221.h, 262
INA3221_REG_CH2_BUSV
 INA3221.h, 262
INA3221_REG_CH2_CRIT_ALERT_LIM
 INA3221.h, 262
INA3221_REG_CH2_SHUNTV
 INA3221.h, 262
INA3221_REG_CH2_WARNING_ALERT_LIM
 INA3221.h, 262
INA3221_REG_CH3_BUSV
 INA3221.h, 262
INA3221_REG_CH3_CRIT_ALERT_LIM
 INA3221.h, 262
INA3221_REG_CH3_SHUNTV
 INA3221.h, 262
INA3221_REG_CH3_WARNING_ALERT_LIM
 INA3221.h, 262
INA3221_REG_CONF
 INA3221.h, 262
INA3221_REG_CONF_AVG_1
 INA3221.h, 262
INA3221_REG_CONF_AVG_1024
 INA3221.h, 262
INA3221_REG_CONF_AVG_128
 INA3221.h, 262
INA3221_REG_CONF_AVG_16
 INA3221.h, 262
INA3221_REG_CONF_AVG_256
 INA3221.h, 262
INA3221_REG_CONF_AVG_4
 INA3221.h, 262
INA3221_REG_CONF_AVG_512
 INA3221.h, 262
INA3221_REG_CONF_AVG_64
 INA3221.h, 262
INA3221_REG_DIE_ID
 INA3221.h, 262
INA3221_REG_MANUF_ID
 INA3221.h, 262
INA3221_REG_MASK_ENABLE
 INA3221.h, 262
INA3221_REG_PWR_VALID_HI_LIM
 INA3221.h, 262
INA3221_REG_PWR_VALID_LO_LIM
 INA3221.h, 262
INA3221_REG_SHUNTV_SUM
 INA3221.h, 262
INA3221_REG_SHUNTV_SUM_LIM
 INA3221.h, 262
ina3221_reg_t
 INA3221.h, 261
includes.h, 186
INFO
 utils.h, 308
init
 BH1750Wrapper, 100
 BME280, 108
 BME280Wrapper, 120
 Event Management, 56
 ISensor, 144
 Telemetry Manager, 90
init_modules
 main.cpp, 313
init_pico_hw

main.cpp, 312
initialize
 Power Management, 70
initialize_radio
 communication.cpp, 213
 communication.h, 216
initialized_
 BH1750Wrapper, 102
 BME280, 114
 BME280Wrapper, 122
 PowerManager, 157
Interface
 Protocol, 38
INTERNAL_FAIL_TO_READ
 Protocol, 37
internal_temperature_command_id
 clock_commands.cpp, 196
INVALID_FORMAT
 Protocol, 37
INVALID_OPERATION
 Protocol, 37
INVALID_VALUE
 Protocol, 37
is_bootloader_reset_pending
 SystemStateManager, 166
is_env_sensor_init_ok
 SystemStateManager, 171
is_gps_collection_paused
 SystemStateManager, 167
is_initialized
 BH1750Wrapper, 101
 BME280Wrapper, 121
 ISensor, 144
is_light_sensor_init_ok
 SystemStateManager, 170
is_radio_init_ok
 SystemStateManager, 170
is_sd_card_mounted
 SystemStateManager, 168
is_telemetry_collection_time
 Telemetry Manager, 93
is_telemetry_flush_time
 Telemetry Manager, 93
ISensor, 143
 ~ISensor, 144
 configure, 145
 get_address, 145
 get_type, 145
 init, 144
 is_initialized, 144
 read_data, 144
last_events_command_id
 Event Commands, 27
last_sensor_command_id
 telemetry_commands.cpp, 211
last_sensor_record_copy
 TelemetryManager, 179
last_sync_time_
 DS3231, 128
last_telemetry_command_id
 telemetry_commands.cpp, 211
last_telemetry_record_copy
 TelemetryManager, 179
lat_dir
 TelemetryRecord, 182
latitude
 TelemetryRecord, 182
lib/clock/DS3231.cpp, 187, 188
lib/clock/DS3231.h, 190, 194
lib/comms/commands/clock_commands.cpp, 195, 196
lib/comms/commands/commands.cpp, 198, 199
lib/comms/commands/commands.h, 199, 201
lib/comms/commands/diagnostic_commands.cpp, 202, 203
lib/comms/commands/event_commands.cpp, 205, 206
lib/comms/commands/gps_commands.cpp, 207, 208
lib/comms/commands/telemetry_commands.cpp, 210, 211
 lib/comms/communication.cpp, 212, 214
lib/comms/communication.h, 214, 220
lib/comms/frame.cpp, 221, 222
lib/comms/protocol.h, 224, 226
lib/comms/receive.cpp, 227, 231
lib/comms/send.cpp, 232, 235
lib/comms/utils_converters.cpp, 235, 236
lib/eventman/event_manager.cpp, 237
lib/eventman/event_manager.h, 239, 242
lib/location/gps_collector.cpp, 243, 244
lib/location/gps_collector.h, 245, 247
lib/location/NMEA/NMEA_data.h, 247, 248
lib/pin_config.h, 249, 256
lib/powerman/INA3221/INA3221.cpp, 256, 257
lib/powerman/INA3221/INA3221.h, 259, 263
lib/powerman/PowerManager.cpp, 265, 266
lib/powerman/PowerManager.h, 267, 268
lib/sensors/BH1750/BH1750.cpp, 269
lib/sensors/BH1750/BH1750.h, 270, 272
lib/sensors/BH1750/BH1750_WRAPPER.cpp, 272, 273
lib/sensors/BH1750/BH1750_WRAPPER.h, 274, 275
lib/sensors/BME280/BME280.cpp, 275, 276
lib/sensors/BME280/BME280.h, 279, 280
lib/sensors/BME280/BME280_WRAPPER.cpp, 282
lib/sensors/BME280/BME280_WRAPPER.h, 283, 284
lib/sensors/ISensor.cpp, 284, 285
lib/sensors/ISensor.h, 286, 287
lib/storage/storage.cpp, 288
lib/storage/storage.h, 289, 290
lib/system_state_manager.h, 291, 292
lib/telemetry/telemetry_manager.cpp, 293, 294
lib/telemetry/telemetry_manager.h, 298, 299
lib/utils.cpp, 301, 306
lib/utils.h, 306, 310
LIGHT
 Sensors, 80
light
 SensorDataRecord, 159

LIGHT_LEVEL
 Sensors, 81

light_sensor_init_status
 SystemStateManager, 174

Location, 58
 collect_gps_data, 60
 MAX_RAW_DATA_LENGTH, 59
 splitString, 59

LOCK
 Event Management, 55

log_event
 Event Management, 56

LOG_FILENAME
 main.cpp, 311

lon_dir
 TelemetryRecord, 182

longitude
 TelemetryRecord, 182

LORA
 Protocol, 38

lora_address_local
 pin_config.h, 255

lora_address_remote
 pin_config.h, 255

lora_cs_pin
 pin_config.h, 255

LORA_DEFAULT_DIO0_PIN
 pin_config.h, 254

LORA_DEFAULT_RESET_PIN
 pin_config.h, 254

LORA_DEFAULT_SPI
 pin_config.h, 254

LORA_DEFAULT_SPI_FREQUENCY
 pin_config.h, 254

LORA_DEFAULT_SS_PIN
 pin_config.h, 254

lora_irq_pin
 pin_config.h, 255

lora_reset_pin
 pin_config.h, 255

lora_tx_done_callback
 communication.cpp, 213
 communication.h, 216

LOST
 Event Management, 55

main
 main.cpp, 314

main.cpp, 311
 core1_entry, 312
 define_system_operating_mode, 313
 init_modules, 313
 init_pico_hw, 312

LOG_FILENAME, 311
 main, 314

MAIN_I2C_PORT
 pin_config.h, 250

MAIN_I2C_SCL_PIN
 pin_config.h, 251

MAIN_I2C_SDA_PIN
 pin_config.h, 251

MAX_PACKET_SIZE
 receive.cpp, 227

MAX_RAW_DATA_LENGTH
 Location, 59

Measurement Functions, 66
 get_current_ma, 67
 get_shunt_voltage, 66
 get_voltage, 68

MILIAMP
 Protocol, 38

minutes
 ds3231_data_t, 129

Mode
 Types, 79

mode_bus_en
 INA3221::conf_reg_t, 123

mode_continuous_en
 INA3221::conf_reg_t, 123

mode_shunt_en
 INA3221::conf_reg_t, 123

MONDAY
 DS3231.h, 193

month
 ds3231_data_t, 129

MSG RECEIVED
 Event Management, 55

MSG SENT
 Event Management, 55

mutex_
 SystemStateManager, 174

nextEventId
 EventManager, 137

NMEAData, 148
 get_gga_tokens, 152
 get_instance, 150
 get_rmc_tokens, 152
 gga_mutex_, 153
 gga_tokens_, 152
 NMEAData, 149
 operator=, 150
 rmc_mutex_, 153
 rmc_tokens_, 152
 update_gga_tokens, 151
 update_rmc_tokens, 151

NONE
 Sensors, 80, 81

NORMAL_MODE
 BME280, 107

NOT_ALLOWED
 Protocol, 37

NUM_CALIB_PARAMS
 BME280, 107

NUM_HUM_CALIB_PARAMS
 BME280, 107

on_receive

communication.h, 217
 receive.cpp, 229
ONE_TIME_HIGH_RES_MODE
 Types, 79
ONE_TIME_HIGH_RES_MODE_2
 Types, 79
ONE_TIME_LOW_RES_MODE
 Types, 79
operation_type_to_string
 Utility Converters, 40
OperationType
 Protocol, 37
operationType
 Frame, 138
operator=
 DS3231, 127
 EventManager, 135
 NMEAData, 150
 PowerManager, 156
 SystemStateManager, 165
OSR_X0
 BME280, 105
OSR_X1
 BME280, 105
OSR_X16
 BME280, 105
OSR_X2
 BME280, 105
OSR_X4
 BME280, 105
OSR_X8
 BME280, 105
Oversampling
 BME280, 105
PA_OUTPUT_PA_BOOST_PIN
 pin_config.h, 254
PA_OUTPUT_RFO_PIN
 pin_config.h, 254
PARAM_INVALID
 Protocol, 37
PARAM_REQUIRED
 Protocol, 37
PARAM_UNNECESSARY
 Protocol, 37
PASS_THROUGH_END
 Event Management, 55
PASS_THROUGH_START
 Event Management, 55
passthrough_command_id
 gps_commands.cpp, 208
pending_bootloader_reset
 SystemStateManager, 173
pin_config.h
 BUFFER_SIZE, 252
 DEBUG_UART_BAUD_RATE, 250
 DEBUG_UART_PORT, 250
 DEBUG_UART_RX_PIN, 250
 DEBUG_UART_TX_PIN, 250
GPS_POWER_ENABLE_PIN, 251
GPS_UART_BAUD_RATE, 251
GPS_UART_PORT, 251
GPS_UART_RX_PIN, 251
GPS_UART_TX_PIN, 251
lora_address_local, 255
lora_address_remote, 255
lora_cs_pin, 255
LORA_DEFAULT_DIO0_PIN, 254
LORA_DEFAULT_RESET_PIN, 254
LORA_DEFAULT_SPI, 254
LORA_DEFAULT_SPI_FREQUENCY, 254
LORA_DEFAULT_SS_PIN, 254
lora_irq_pin, 255
lora_reset_pin, 255
MAIN_I2C_PORT, 250
MAIN_I2C_SCL_PIN, 251
MAIN_I2C_SDA_PIN, 251
PA_OUTPUT_PA_BOOST_PIN, 254
PA_OUTPUT_RFO_PIN, 254
READ_BIT, 253
SD_CARD_DETECT_PIN, 253
SD_CS_PIN, 253
SD_MISO_PIN, 252
SD_MOSI_PIN, 252
SD_SCK_PIN, 252
SD_SPI_PORT, 252
SENSORS_I2C_PORT, 252
SENSORS_I2C_SCL_PIN, 252
SENSORS_I2C_SDA_PIN, 252
SENSORS_POWER_ENABLE_PIN, 251
SPI_PORT, 253
SX1278_CS, 253
SX1278_MISO, 253
SX1278_MOSI, 253
SX1278_SCK, 253
POWER
 Event Management, 54
Power Management, 68
 configure, 73
 get_current_charge_solar, 72
 get_current_charge_total, 73
 get_current_charge_usb, 72
 get_current_draw, 72
 get_instance, 69
 get_voltage_5v, 71
 get_voltage_battery, 71
 get_voltage_solar, 71
 initialize, 70
 PowerManager, 69
 read_device_ids, 70
POWER_FALLING
 Event Management, 54
power_mode_command_id
 Diagnostic Commands, 24
POWER_OFF
 Event Management, 55
POWER_ON

Event Management, 55
Types, 79
power_status_command_id
 gps_commands.cpp, 208
PowerEvent
 Event Management, 54
powerman_mutex_
 PowerManager, 157
PowerManager, 153
 BATTERY_FULL_THRESHOLD, 157
 BATTERY_LOW_THRESHOLD, 157
 ina3221_, 157
 initialized_, 157
 operator=, 156
 Power Management, 69
 powerman_mutex_, 157
 PowerManager, 155
 SOLAR_CURRENT_THRESHOLD, 156
 USB_CURRENT_THRESHOLD, 156
PRESSURE
 Sensors, 81
pressure
 SensorDataRecord, 158
PRESSURE_OVERSAMPLING
 BME280, 107
process_lora_packet
 receive.cpp, 228
Protocol, 36
 BOOL, 38
 CELSIUS, 38
 DATETIME, 38
 ERR, 38
 ErrorCode, 37
 FAIL_TO_SET, 37
 GET, 38
 Interface, 38
 INTERNAL_FAIL_TO_READ, 37
 INVALID_FORMAT, 37
 INVALID_OPERATION, 37
 INVALID_VALUE, 37
 LORA, 38
 MILIAMP, 38
 NOT_ALLOWED, 37
 OperationType, 37
 PARAM_INVALID, 37
 PARAM_REQUIRED, 37
 PARAM_UNNECESSARY, 37
 RES, 38
 SECOND, 38
 SEQ, 38
 SET, 38
 TEXT, 38
 UART, 38
 UNDEFINED, 38
 UNKNOWN_ERROR, 37
 VAL, 38
 ValueUnit, 38
 VOLT, 38
protocol.h
 DELIMITER, 225
 FRAME_BEGIN, 225
 FRAME_END, 225
pwr_valid_alert
 INA3221::masken_reg_t, 146
RADIO_ERROR
 Event Management, 55
RADIO_INIT
 Event Management, 55
radio_init_status
 SystemStateManager, 174
READ_BIT
 pin_config.h, 253
read_data
 BH1750Wrapper, 100
 BME280Wrapper, 120
 ISensor, 144
read_device_ids
 Power Management, 70
read_raw_all
 BME280, 108
read_register
 BME280, 110, 111
 Configuration Functions, 64
read_temperature
 RTC clock, 46
receive.cpp
 extract_and_process_frames, 228
 handle_uart_input, 230
 MAX_PACKET_SIZE, 227
 on_receive, 229
 process_lora_packet, 228
REG_CONFIG
 BME280, 106
REG_CTRL_HUM
 BME280, 106
REG_CTRL_MEAS
 BME280, 106
REG_DIG_H1
 BME280, 106
REG_DIG_H2
 BME280, 106
REG_DIG_H3
 BME280, 106
REG_DIG_H4
 BME280, 106
REG_DIG_H5
 BME280, 106
REG_DIG_H6
 BME280, 106
REG_DIG_P1_LSB
 BME280, 106
REG_DIG_P1_MSB
 BME280, 106
REG_DIG_P2_LSB
 BME280, 106
REG_DIG_P2_MSB

BME280, 106
 REG_DIG_P3_LSB
 BME280, 106
 REG_DIG_P3_MSB
 BME280, 106
 REG_DIG_P4_LSB
 BME280, 106
 REG_DIG_P4_MSB
 BME280, 106
 REG_DIG_P5_LSB
 BME280, 106
 REG_DIG_P5_MSB
 BME280, 106
 REG_DIG_P6_LSB
 BME280, 106
 REG_DIG_P6_MSB
 BME280, 106
 REG_DIG_P7_LSB
 BME280, 106
 REG_DIG_P7_MSB
 BME280, 106
 REG_DIG_P8_LSB
 BME280, 106
 REG_DIG_P8_MSB
 BME280, 106
 REG_DIG_P9_LSB
 BME280, 106
 REG_DIG_P9_MSB
 BME280, 106
 REG_DIG_T1_LSB
 BME280, 106
 REG_DIG_T1_MSB
 BME280, 106
 REG_DIG_T2_LSB
 BME280, 106
 REG_DIG_T2_MSB
 BME280, 106
 REG_DIG_T3_LSB
 BME280, 106
 REG_DIG_T3_MSB
 BME280, 106
 REG_HUMIDITY_MSB
 BME280, 106
 REG_PRESSURE_MSB
 BME280, 106
 REG_RESET
 BME280, 106
 REG_TEMPERATURE_MSB
 BME280, 106
 RES
 Protocol, 38
 reserved
 INA3221::masken_reg_t, 148
 RESET
 Types, 79
 reset
 BME280, 108
 INA3221::conf_reg_t, 124

rmc_mutex_
 NMEAData, 153
 rmc_tokens_
 NMEAData, 152
 RTC clock, 42
 DS3231, 43
 get_instance, 43
 get_local_time, 48
 get_time, 44
 get_timezone_offset, 46
 i2c_read_reg, 49
 i2c_write_reg, 50
 read_temperature, 46
 set_time, 45
 set_timezone_offset, 47

sample_interval_ms
 TelemetryManager, 178

satellites
 TelemetryRecord, 183

SATURDAY
 DS3231.h, 193

save_to_storage
 Event Management, 57

scan_connected_sensors
 SensorWrapper, 161

SD_CARD_DETECT_PIN
 pin_config.h, 253

sd_card_init_status
 SystemStateManager, 173

sd_card_mounted
 SystemStateManager, 173

SD_CS_PIN
 pin_config.h, 253

SD_MISO_PIN
 pin_config.h, 252

SD_MOSI_PIN
 pin_config.h, 252

SD_SCK_PIN
 pin_config.h, 252

SD_SPI_PORT
 pin_config.h, 252

SECOND
 Protocol, 38

seconds
 ds3231_data_t, 129

send.cpp
 send_frame_lora, 233
 send_frame_uart, 234
 send_message, 233

send_frame_lora
 communication.h, 220
 send.cpp, 233

send_frame_uart
 communication.h, 219
 send.cpp, 234

send_message
 communication.h, 218
 send.cpp, 233

sensor_ Configuration Functions, 65
 BH1750Wrapper, 102
 BME280Wrapper, 122
sensor_configure Sensors, 81
sensor_data_buffer TelemetryManager, 179
SENSOR_DATA_CSV_PATH Telemetry Manager, 86
sensor_init Sensors, 81
sensor_read_data Sensors, 82
SensorDataRecord, 158
 humidity, 159
 light, 159
 pressure, 158
 temperature, 158
 timestamp, 158
SensorDataTypeldentifier Sensors, 80
Sensors, 79
 ENVIRONMENT, 80
 get_sensor, 82
 HUMIDITY, 81
 LIGHT, 80
 LIGHT_LEVEL, 81
 NONE, 80, 81
 PRESSURE, 81
 sensor_configure, 81
 sensor_init, 81
 sensor_read_data, 82
 SensorDataTypeldentifier, 80
 SensorType, 80
 TEMPERATURE, 81
sensors SensorWrapper, 162
SENSORS_I2C_PORT pin_config.h, 252
SENSORS_I2C_SCL_PIN pin_config.h, 252
SENSORS_I2C_SDA_PIN pin_config.h, 252
SENSORS_POWER_ENABLE_PIN pin_config.h, 251
SensorType Sensors, 80
SensorWrapper, 159
 get_available_sensors, 161
 get_instance, 161
 scan_connected_sensors, 161
 sensors, 162
 SensorWrapper, 160
SEQ Protocol, 38
SET Protocol, 38
set_averaging_mode

set_bootloader_reset_pending SystemStateManager, 167
set_env_sensor_init_ok SystemStateManager, 171
set_gps_collection_paused SystemStateManager, 167
set_light_sensor_init_ok SystemStateManager, 171
set_mode_continuous Configuration Functions, 64
set_mode_triggered Configuration Functions, 65
set_operating_mode SystemStateManager, 172
set_radio_init_ok SystemStateManager, 170
set_sd_card_mounted SystemStateManager, 168
set_time RTC clock, 45
set_timezone_offset RTC clock, 47
set_uart_verbosity SystemStateManager, 169
shunt_conv_time INA3221::conf_reg_t, 123
shunt_sum_alert INA3221::masken_reg_t, 147
shunt_sum_en_ch1 INA3221::masken_reg_t, 148
shunt_sum_en_ch2 INA3221::masken_reg_t, 148
shunt_sum_en_ch3 INA3221::masken_reg_t, 148
SHUNT_VOLTAGE_LSB_UV INA3221.h, 263
SHUTDOWN Event Management, 54
SILENT utils.h, 308
SOLAR_ACTIVE Event Management, 54
SOLAR_CURRENT_THRESHOLD PowerManager, 156
SOLAR_INACTIVE Event Management, 54
speed TelemetryRecord, 183
SPI_PORT pin_config.h, 253
splitString Location, 59
Storage, 83
 fs_init, 83
 fs_stop, 84
string_to_operation_type Utility Converters, 40

SUNDAY
 DS3231.h, 193

SX1278_CS
 pin_config.h, 253

SX1278_MISO
 pin_config.h, 253

SX1278_MOSI
 pin_config.h, 253

SX1278_SCK
 pin_config.h, 253

sync_interval_minutes_
 DS3231, 128

SYSTEM
 Event Management, 54

System State Manager, 84
 BATTERY_POWERED, 85
 SystemOperatingMode, 85
 USB_POWERED, 85

system_operating_mode
 SystemStateManager, 174

system_voltage
 TelemetryRecord, 181

SystemEvent
 Event Management, 54

SystemOperatingMode
 System State Manager, 85

SystemStateManager, 162
 env_sensor_init_status, 174
 get_instance, 165
 get_operating_mode, 172
 get_uart_verbosity, 169
 gps_collection_paused, 173
 is_bootloader_reset_pending, 166
 is_env_sensor_init_ok, 171
 is_gps_collection_paused, 167
 is_light_sensor_init_ok, 170
 is_radio_init_ok, 170
 is_sd_card_mounted, 168
 light_sensor_init_status, 174
 mutex_, 174
 operator=, 165
 pending_bootloader_reset, 173
 radio_init_status, 174
 sd_card_init_status, 173
 sd_card_mounted, 173
 set_bootloader_reset_pending, 167
 set_env_sensor_init_ok, 171
 set_gps_collection_paused, 167
 set_light_sensor_init_ok, 171
 set_operating_mode, 172
 set_radio_init_ok, 170
 set_sd_card_mounted, 168
 set_uart_verbosity, 169
 system_operating_mode, 174

SystemStateManager, 164
 uart_verbosity, 173

t_fine
 BME280, 114

Telemetry Buffer Commands, 30
 handle_get_last_sensor_record, 31
 handle_get_last_telemetry_record, 30

Telemetry Manager, 85
 collect_gps_telemetry, 89
 collect_power_telemetry, 87
 collect_sensor_telemetry, 89
 collect_telemetry, 91
 DEFAULT_FLUSH_THRESHOLD, 87
 DEFAULT_SAMPLE_INTERVAL_MS, 86
 emit_power_events, 88
 flush_telemetry, 92
 get_last_sensor_record_csv, 94
 get_last_telemetry_record_csv, 94
 init, 90
 is_telemetry_collection_time, 93
 is_telemetry_flush_time, 93
 SENSOR_DATA_CSV_PATH, 86
 TELEMETRY_CSV_PATH, 86
 TelemetryManager, 90
 to_csv, 87

telemetry_buffer
 TelemetryManager, 179

telemetry_buffer_count
 TelemetryManager, 179

TELEMETRY_BUFFER_SIZE
 TelemetryManager, 178

telemetry_buffer_write_index
 TelemetryManager, 179

telemetry_commands.cpp
 last_sensor_command_id, 211
 last_telemetry_command_id, 211
 telemetry_commands_group, 211

telemetry_commands_group
 telemetry_commands.cpp, 211

TELEMETRY_CSV_PATH
 Telemetry Manager, 86

telemetry_mutex
 TelemetryManager, 180

TelemetryManager, 175
 ~TelemetryManager, 177
 DEFAULT_FLUSH_THRESHOLD, 178
 DEFAULT_SAMPLE_INTERVAL_MS, 178
 flush_sensor_data, 177
 flush_threshold, 179
 get_instance, 177
 get_telemetry_buffer_count, 178
 get_telemetry_buffer_write_index, 178
 last_sensor_record_copy, 179
 last_telemetry_record_copy, 179
 sample_interval_ms, 178
 sensor_data_buffer, 179
 Telemetry Manager, 90
 telemetry_buffer, 179
 telemetry_buffer_count, 179
 TELEMETRY_BUFFER_SIZE, 178
 telemetry_buffer_write_index, 179
 telemetry_mutex, 180

TelemetryRecord, 180
altitude, 183
battery_voltage, 181
build_version, 181
charge_current_solar, 181
charge_current_usb, 181
course, 183
date, 183
discharge_current, 182
fix_quality, 183
lat_dir, 182
latitude, 182
lon_dir, 182
longitude, 182
satellites, 183
speed, 183
system_voltage, 181
time, 182
timestamp, 181

TEMPERATURE
Sensors, 81

temperature
SensorDataRecord, 158

TEMPERATURE_OVERSAMPLING
BME280, 107

TEXT
Protocol, 38

THURSDAY
DS3231.h, 193

time
TelemetryRecord, 182

time_command_id
clock_commands.cpp, 195

timestamp
event_manager.h, 241
EventLog, 132
SensorDataRecord, 158
TelemetryRecord, 181

timezone_offset_command_id
clock_commands.cpp, 196

timezone_offset_minutes_
DS3231, 128

timing_ctrl_alert
INA3221::masken_reg_t, 146

to_csv
Telemetry Manager, 87

TRIGGERED_MODE
INA3221.h, 263

TUESDAY
DS3231.h, 193

Types, 78
CONTINUOUS_HIGH_RES_MODE, 79
CONTINUOUS_HIGH_RES_MODE_2, 79
CONTINUOUS_LOW_RES_MODE, 79
Mode, 79
ONE_TIME_HIGH_RES_MODE, 79
ONE_TIME_HIGH_RES_MODE_2, 79
ONE_TIME_LOW_RES_MODE, 79

POWER_ON, 79
RESET, 79
UNCONFIGURED_POWER_DOWN, 79

UART
Protocol, 38

UART_ERROR
Event Management, 55

uart_print
utils.cpp, 304
utils.h, 309

uart_verbosity
SystemStateManager, 173

UNCONFIGURED_POWER_DOWN
Types, 79

UNDEFINED
Protocol, 38

unit
Frame, 139

UNKNOWN_ERROR
Protocol, 37

update_gga_tokens
NMEAData, 151

update_rmc_tokens
NMEAData, 151

uptime_command_id
Diagnostic Commands, 25

USB_CONNECTED
Event Management, 54

USB_CURRENT_THRESHOLD
PowerManager, 156

USB_DISCONNECTED
Event Management, 54

USB_POWERED
System State Manager, 85

Utility Converters, 39
error_code_to_string, 41
operation_type_to_string, 40
string_to_operation_type, 40
value_unit_type_to_string, 39

utils.cpp
get_level_color, 302
get_level_prefix, 303
uart_print, 304

utils.h
ANSI_BLUE, 308
ANSI_GREEN, 308
ANSI_RED, 308
ANSI_RESET, 308
ANSI_YELLOW, 308
DEBUG, 308
ERROR, 308
INFO, 308
SILENT, 308
uart_print, 309
VerbosityLevel, 308
WARNING, 308

VAL

Protocol, 38
value
 Frame, 139
value_unit_type_to_string
 Utility Converters, 39
ValueUnit
 Protocol, 38
verbosity_command_id
 Diagnostic Commands, 25
VerbosityLevel
 utils.h, 308
VOLT
 Protocol, 38

warn_alert_ch1
 INA3221::masken_reg_t, 147
warn_alert_ch2
 INA3221::masken_reg_t, 147
warn_alert_ch3
 INA3221::masken_reg_t, 146
warn_alert_latch_en
 INA3221::masken_reg_t, 147
WARNING
 utils.h, 308
WATCHDOG_RESET
 Event Management, 54
WEDNESDAY
 DS3231.h, 193
write8
 BH1750 Light Sensor, 76
write_register
 BME280, 110
writeIndex
 EventManager, 137

year
 ds3231_data_t, 130