

KubiSat Firmware

Generated by Doxygen 1.13.2

1 Clock Commands	1
2 Topic Index	3
2.1 Topics	3
3 Hierarchical Index	5
3.1 Class Hierarchy	5
4 Class Index	7
4.1 Class List	7
5 File Index	9
5.1 File List	9
6 Topic Documentation	13
6.1 Clock Management Commands	13
6.1.1 Detailed Description	13
6.1.2 Function Documentation	13
6.1.2.1 handle_time()	13
6.1.2.2 handle_timezone_offset()	14
6.1.2.3 handle_clock_sync_interval()	15
6.1.2.4 handle_get_last_sync_time()	16
6.1.3 Variable Documentation	17
6.1.3.1 systemClock	17
6.2 Command System	17
6.2.1 Detailed Description	17
6.2.2 Typedef Documentation	17
6.2.2.1 CommandHandler	17
6.2.2.2 CommandMap	18
6.2.3 Function Documentation	18
6.2.3.1 execute_command()	18
6.2.4 Variable Documentation	19
6.2.4.1 command_handlers	19
6.3 Diagnostic Commands	19
6.3.1 Detailed Description	20
6.3.2 Function Documentation	20
6.3.2.1 handle_get_commands_list()	20
6.3.2.2 handle_get_build_version()	21
6.3.2.3 handle_verbosity()	22
6.3.2.4 handle_enter_bootloader_mode()	24
6.4 Event Commands	25
6.4.1 Detailed Description	25
6.4.2 Function Documentation	25
6.4.2.1 handle_get_last_events()	25

6.4.2.2 handle_get_event_count()	26
6.5 GPS Commands	27
6.5.1 Detailed Description	27
6.5.2 Function Documentation	27
6.5.2.1 handle_gps_power_status()	27
6.5.2.2 handle_enable_gps_uart_passthrough()	28
6.5.2.3 handle_get_rmc_data()	29
6.5.2.4 handle_get_gga_data()	30
6.6 Power Commands	31
6.6.1 Detailed Description	31
6.6.2 Function Documentation	31
6.6.2.1 handle_get_power_manager_ids()	31
6.6.2.2 handle_get_voltage_battery()	32
6.6.2.3 handle_get_voltage_5v()	33
6.6.2.4 handle_get_current_charge_usb()	34
6.6.2.5 handle_get_current_charge_solar()	35
6.6.2.6 handle_get_current_charge_total()	35
6.6.2.7 handle_get_current_draw()	36
6.7 Sensor Commands	37
6.7.1 Detailed Description	37
6.7.2 Function Documentation	37
6.7.2.1 handle_get_sensor_data()	37
6.7.2.2 handle_sensor_config()	38
6.7.2.3 handle_get_sensor_list()	39
6.8 Storage Commands	40
6.8.1 Detailed Description	41
6.8.2 Function Documentation	41
6.8.2.1 handle_list_files()	41
6.8.2.2 handle_mount()	41
6.9 Telemetry Buffer Commands	42
6.9.1 Detailed Description	43
6.9.2 Function Documentation	43
6.9.2.1 handle_get_last_telemetry_record()	43
6.9.2.2 handle_get_last_sensor_record()	44
6.10 Frame Handling	44
6.10.1 Detailed Description	45
6.10.2 Function Documentation	45
6.10.2.1 frame_encode()	45
6.10.2.2 frame_decode()	46
6.10.2.3 frame_process()	47
6.10.2.4 frame_build()	48
6.11 Event Manager	49

6.11.1 Detailed Description	51
6.11.2 Enumeration Type Documentation	51
6.11.2.1 EventGroup	51
6.11.2.2 SystemEvent	51
6.11.2.3 PowerEvent	52
6.11.2.4 CommsEvent	52
6.11.2.5 GPSEvent	52
6.11.2.6 ClockEvent	53
6.11.3 Function Documentation	53
6.11.3.1 __attribute__()	53
6.11.3.2 log_event()	53
6.11.3.3 get_event()	54
6.11.4 Variable Documentation	55
6.11.4.1 eventLogId	55
6.11.4.2 systemClock	55
6.11.4.3 eventManager [1/2]	55
6.11.4.4 __attribute__	55
6.11.4.5 eventManager [2/2]	55
6.12 INA3221 Power Monitor	56
6.12.1 Detailed Description	56
6.12.2 Configuration Functions	56
6.12.2.1 Detailed Description	57
6.12.2.2 Function Documentation	57
6.12.3 Measurement Functions	66
6.12.3.1 Detailed Description	66
6.12.3.2 Function Documentation	66
6.12.4 Alert Functions	68
6.12.4.1 Detailed Description	69
6.12.4.2 Function Documentation	69
6.13 Telemetry Manager	74
6.13.1 Detailed Description	75
6.13.2 Function Documentation	75
6.13.2.1 telemetry_init()	75
6.13.2.2 collect_telemetry()	76
6.13.2.3 flush_telemetry()	77
6.13.2.4 flush_sensor_data()	77
6.13.2.5 is_telemetry_collection_time()	78
6.13.2.6 is_telemetry_flush_time()	79
6.13.2.7 get_telemetry_sample_interval()	80
6.13.2.8 set_telemetry_sample_interval()	80
6.13.2.9 get_telemetry_flush_threshold()	80
6.13.2.10 set_telemetry_flush_threshold()	80

6.13.2.11 get_last_telemetry_record_csv()	81
6.13.2.12 get_last_sensor_record_csv()	81
6.13.3 Variable Documentation	82
6.13.3.1 TELEMETRY_BUFFER_SIZE [1/2]	82
6.13.3.2 telemetry_buffer	82
6.13.3.3 telemetry_buffer_count	82
6.13.3.4 telemetry_buffer_write_index	83
6.13.3.5 sensor_data_buffer	83
6.13.3.6 sensor_data_buffer_count	83
6.13.3.7 sensor_data_buffer_write_index	83
6.13.3.8 telemetry_mutex	83
6.13.3.9 TELEMETRY_BUFFER_SIZE [2/2]	83
7 Class Documentation	85
7.1 BH1750 Class Reference	85
7.1.1 Detailed Description	85
7.1.2 Member Enumeration Documentation	85
7.1.2.1 Mode	85
7.1.3 Constructor & Destructor Documentation	86
7.1.3.1 BH1750()	86
7.1.4 Member Function Documentation	86
7.1.4.1 begin()	86
7.1.4.2 configure()	87
7.1.4.3 get_light_level()	87
7.1.4.4 write8()	87
7.1.5 Member Data Documentation	88
7.1.5.1 _i2c_addr	88
7.2 BH1750Wrapper Class Reference	88
7.2.1 Detailed Description	89
7.2.2 Constructor & Destructor Documentation	89
7.2.2.1 BH1750Wrapper()	89
7.2.3 Member Function Documentation	89
7.2.3.1 get_i2c_addr()	89
7.2.3.2 init()	89
7.2.3.3 read_data()	90
7.2.3.4 is_initialized()	90
7.2.3.5 get_type()	90
7.2.3.6 configure()	90
7.2.3.7 get_address()	90
7.2.4 Member Data Documentation	90
7.2.4.1 sensor_	90
7.2.4.2 initialized_	91

7.3 BME280 Class Reference	91
7.3.1 Detailed Description	92
7.3.2 Constructor & Destructor Documentation	93
7.3.2.1 BME280()	93
7.3.3 Member Function Documentation	93
7.3.3.1 init()	93
7.3.3.2 reset()	93
7.3.3.3 read_raw_all()	93
7.3.3.4 convert_temperature()	94
7.3.3.5 convert_pressure()	94
7.3.3.6 convert_humidity()	94
7.3.3.7 configure_sensor()	94
7.3.3.8 get_calibration_parameters()	94
7.3.4 Member Data Documentation	95
7.3.4.1 ADDR_SDO_LOW	95
7.3.4.2 ADDR_SDO_HIGH	95
7.3.4.3 i2c_port	95
7.3.4.4 device_addr	95
7.3.4.5 calib_params	95
7.3.4.6 initialized_	95
7.3.4.7 t_fine	95
7.3.4.8 REG_CONFIG	96
7.3.4.9 REG_CTRL_MEAS	96
7.3.4.10 REG_CTRL_HUM	96
7.3.4.11 REG_RESET	96
7.3.4.12 REG_PRESSURE_MSB	96
7.3.4.13 REG_TEMPERATURE_MSB	96
7.3.4.14 REG_HUMIDITY_MSB	96
7.3.4.15 REG_DIG_T1_LSB	96
7.3.4.16 REG_DIG_T1_MSB	97
7.3.4.17 REG_DIG_T2_LSB	97
7.3.4.18 REG_DIG_T2_MSB	97
7.3.4.19 REG_DIG_T3_LSB	97
7.3.4.20 REG_DIG_T3_MSB	97
7.3.4.21 REG_DIG_P1_LSB	97
7.3.4.22 REG_DIG_P1_MSB	97
7.3.4.23 REG_DIG_P2_LSB	97
7.3.4.24 REG_DIG_P2_MSB	98
7.3.4.25 REG_DIG_P3_LSB	98
7.3.4.26 REG_DIG_P3_MSB	98
7.3.4.27 REG_DIG_P4_LSB	98
7.3.4.28 REG_DIG_P4_MSB	98

7.3.4.29 REG_DIG_P5_LSB	98
7.3.4.30 REG_DIG_P5_MSB	98
7.3.4.31 REG_DIG_P6_LSB	98
7.3.4.32 REG_DIG_P6_MSB	99
7.3.4.33 REG_DIG_P7_LSB	99
7.3.4.34 REG_DIG_P7_MSB	99
7.3.4.35 REG_DIG_P8_LSB	99
7.3.4.36 REG_DIG_P8_MSB	99
7.3.4.37 REG_DIG_P9_LSB	99
7.3.4.38 REG_DIG_P9_MSB	99
7.3.4.39 REG_DIG_H1	99
7.3.4.40 REG_DIG_H2	100
7.3.4.41 REG_DIG_H3	100
7.3.4.42 REG_DIG_H4	100
7.3.4.43 REG_DIG_H5	100
7.3.4.44 REG_DIG_H6	100
7.3.4.45 NUM_CALIB_PARAMS	100
7.4 BME280CalibParam Struct Reference	100
7.4.1 Detailed Description	101
7.4.2 Member Data Documentation	101
7.4.2.1 dig_t1	101
7.4.2.2 dig_t2	101
7.4.2.3 dig_t3	101
7.4.2.4 dig_p1	102
7.4.2.5 dig_p2	102
7.4.2.6 dig_p3	102
7.4.2.7 dig_p4	102
7.4.2.8 dig_p5	102
7.4.2.9 dig_p6	102
7.4.2.10 dig_p7	102
7.4.2.11 dig_p8	102
7.4.2.12 dig_p9	103
7.4.2.13 dig_h1	103
7.4.2.14 dig_h2	103
7.4.2.15 dig_h3	103
7.4.2.16 dig_h4	103
7.4.2.17 dig_h5	103
7.4.2.18 dig_h6	103
7.5 BME280Wrapper Class Reference	104
7.5.1 Detailed Description	105
7.5.2 Constructor & Destructor Documentation	105
7.5.2.1 BME280Wrapper()	105

7.5.3 Member Function Documentation	105
7.5.3.1 init()	105
7.5.3.2 read_data()	105
7.5.3.3 is_initialized()	105
7.5.3.4 get_type()	106
7.5.3.5 configure()	106
7.5.3.6 get_address()	106
7.5.4 Member Data Documentation	106
7.5.4.1 sensor_	106
7.5.4.2 initialized_	106
7.6 INA3221::conf_reg_t Struct Reference	106
7.6.1 Detailed Description	107
7.6.2 Member Data Documentation	107
7.6.2.1 mode_shunt_en	107
7.6.2.2 mode_bus_en	107
7.6.2.3 mode_continious_en	107
7.6.2.4 shunt_conv_time	107
7.6.2.5 bus_conv_time	108
7.6.2.6 avg_mode	108
7.6.2.7 ch3_en	108
7.6.2.8 ch2_en	108
7.6.2.9 ch1_en	108
7.6.2.10 reset	108
7.7 DS3231 Class Reference	108
7.7.1 Detailed Description	110
7.7.2 Constructor & Destructor Documentation	110
7.7.2.1 DS3231()	110
7.7.3 Member Function Documentation	110
7.7.3.1 set_time()	110
7.7.3.2 get_time()	111
7.7.3.3 read_temperature()	112
7.7.3.4 set_unix_time()	113
7.7.3.5 get_unix_time()	114
7.7.3.6 clock_enable()	115
7.7.3.7 get_timezone_offset()	116
7.7.3.8 set_timezone_offset()	116
7.7.3.9 get_clock_sync_interval()	117
7.7.3.10 set_clock_sync_interval()	117
7.7.3.11 get_last_sync_time()	118
7.7.3.12 update_last_sync_time()	119
7.7.3.13 get_local_time()	119
7.7.3.14 is_sync_needed()	120

7.7.3.15 sync_clock_with_gps()	120
7.7.3.16 i2c_read_reg()	121
7.7.3.17 i2c_write_reg()	122
7.7.3.18 bin_to_bcd()	124
7.7.3.19 bcd_to_bin()	124
7.7.4 Member Data Documentation	125
7.7.4.1 i2c	125
7.7.4.2 ds3231_addr	125
7.7.4.3 clock_mutex_	126
7.7.4.4 timezone_offset_minutes_	126
7.7.4.5 sync_interval_minutes_	126
7.7.4.6 last_sync_time_	126
7.8 ds3231_data_t Struct Reference	126
7.8.1 Detailed Description	127
7.8.2 Member Data Documentation	127
7.8.2.1 seconds	127
7.8.2.2 minutes	127
7.8.2.3 hours	127
7.8.2.4 day	128
7.8.2.5 date	128
7.8.2.6 month	128
7.8.2.7 year	128
7.8.2.8 century	128
7.9 EventEmitter Class Reference	128
7.9.1 Detailed Description	129
7.9.2 Member Function Documentation	129
7.9.2.1 emit()	129
7.10 EventLog Class Reference	130
7.10.1 Detailed Description	131
7.10.2 Member Function Documentation	131
7.10.2.1 to_string()	131
7.10.3 Member Data Documentation	131
7.10.3.1 id	131
7.10.3.2 timestamp	131
7.10.3.3 group	132
7.10.3.4 event	132
7.11 EventManager Class Reference	132
7.11.1 Detailed Description	133
7.11.2 Constructor & Destructor Documentation	133
7.11.2.1 EventManager()	133
7.11.2.2 ~EventManager()	134
7.11.3 Member Function Documentation	134

7.11.3.1 init()	134
7.11.3.2 get_event_count()	134
7.11.3.3 save_to_storage()	135
7.11.3.4 load_from_storage()	135
7.11.4 Member Data Documentation	135
7.11.4.1 events	135
7.11.4.2 eventCount	136
7.11.4.3 writeIndex	136
7.11.4.4 eventMutex	136
7.11.4.5 nextEventId	136
7.11.4.6 eventsSinceFlush	136
7.12 EventManagerImpl Class Reference	137
7.12.1 Detailed Description	138
7.12.2 Constructor & Destructor Documentation	139
7.12.2.1 EventManagerImpl()	139
7.12.3 Member Function Documentation	139
7.12.3.1 save_to_storage()	139
7.12.3.2 load_from_storage()	140
7.13 FileHandle Struct Reference	140
7.13.1 Detailed Description	140
7.13.2 Member Data Documentation	140
7.13.2.1 fd	140
7.13.2.2 is_open	140
7.14 Frame Struct Reference	141
7.14.1 Detailed Description	141
7.14.2 Member Data Documentation	142
7.14.2.1 header	142
7.14.2.2 direction	142
7.14.2.3 operationType	142
7.14.2.4 group	142
7.14.2.5 command	143
7.14.2.6 value	143
7.14.2.7 unit	143
7.14.2.8 footer	143
7.15 HMC5883L Class Reference	143
7.15.1 Detailed Description	144
7.15.2 Constructor & Destructor Documentation	144
7.15.2.1 HMC5883L()	144
7.15.3 Member Function Documentation	144
7.15.3.1 init()	144
7.15.3.2 read()	145
7.15.3.3 write_register()	145

7.15.3.4 <code>read_register()</code>	145
7.15.4 Member Data Documentation	146
7.15.4.1 <code>i2c</code>	146
7.15.4.2 <code>address</code>	146
7.16 HMC5883LWrapper Class Reference	146
7.16.1 Detailed Description	147
7.16.2 Constructor & Destructor Documentation	147
7.16.2.1 <code>HMC5883LWrapper()</code>	147
7.16.3 Member Function Documentation	148
7.16.3.1 <code>init()</code>	148
7.16.3.2 <code>read_data()</code>	148
7.16.3.3 <code>is_initialized()</code>	148
7.16.3.4 <code>get_type()</code>	148
7.16.3.5 <code>configure()</code>	148
7.16.3.6 <code>get_address()</code>	148
7.16.4 Member Data Documentation	149
7.16.4.1 <code>sensor_</code>	149
7.16.4.2 <code>initialized_</code>	149
7.17 INA3221 Class Reference	149
7.17.1 Detailed Description	151
7.17.2 Member Function Documentation	151
7.17.2.1 <code>_read()</code>	151
7.17.2.2 <code>_write()</code>	153
7.17.2.3 <code>get_current()</code>	154
7.17.3 Member Data Documentation	155
7.17.3.1 <code>_i2c_addr</code>	155
7.17.3.2 <code>_i2c</code>	155
7.17.3.3 <code>_shuntRes</code>	155
7.17.3.4 <code>_filterRes</code>	155
7.17.3.5 <code>_masken_reg</code>	155
7.18 ISensor Class Reference	155
7.18.1 Detailed Description	156
7.18.2 Constructor & Destructor Documentation	156
7.18.2.1 <code>~ISensor()</code>	156
7.18.3 Member Function Documentation	156
7.18.3.1 <code>init()</code>	156
7.18.3.2 <code>read_data()</code>	156
7.18.3.3 <code>is_initialized()</code>	156
7.18.3.4 <code>get_type()</code>	156
7.18.3.5 <code>configure()</code>	157
7.18.3.6 <code>get_address()</code>	157
7.19 INA3221::masken_reg_t Struct Reference	157

7.19.1 Detailed Description	157
7.19.2 Member Data Documentation	158
7.19.2.1 conv_ready	158
7.19.2.2 timing_ctrl_alert	158
7.19.2.3 pwr_valid_alert	158
7.19.2.4 warn_alert_ch3	158
7.19.2.5 warn_alert_ch2	158
7.19.2.6 warn_alert_ch1	158
7.19.2.7 shunt_sum_alert	158
7.19.2.8 crit_alert_ch3	159
7.19.2.9 crit_alert_ch2	159
7.19.2.10 crit_alert_ch1	159
7.19.2.11 crit_alert_latch_en	159
7.19.2.12 warn_alert_latch_en	159
7.19.2.13 shunt_sum_en_ch3	159
7.19.2.14 shunt_sum_en_ch2	159
7.19.2.15 shunt_sum_en_ch1	159
7.19.2.16 reserved	160
7.20 MPU6050Wrapper Class Reference	160
7.20.1 Detailed Description	161
7.20.2 Constructor & Destructor Documentation	161
7.20.2.1 MPU6050Wrapper()	161
7.20.3 Member Function Documentation	161
7.20.3.1 init()	161
7.20.3.2 read_data()	161
7.20.3.3 is_initialized()	161
7.20.3.4 get_type()	161
7.20.3.5 configure()	162
7.20.4 Member Data Documentation	162
7.20.4.1 sensor_	162
7.20.4.2 initialized_	162
7.21 NMEAData Class Reference	162
7.21.1 Detailed Description	163
7.21.2 Constructor & Destructor Documentation	163
7.21.2.1 NMEAData()	163
7.21.3 Member Function Documentation	163
7.21.3.1 update_rmc_tokens()	163
7.21.3.2 update_gga_tokens()	163
7.21.3.3 get_rmc_tokens()	163
7.21.3.4 get_gga_tokens()	163
7.21.3.5 has_valid_time()	164
7.21.3.6 get_unix_time()	164

7.21.4 Member Data Documentation	164
7.21.4.1 rmc_tokens_	164
7.21.4.2 gga_tokens_	164
7.21.4.3 rmc_mutex_	165
7.21.4.4 gga_mutex_	165
7.22 PowerManager Class Reference	165
7.22.1 Detailed Description	166
7.22.2 Constructor & Destructor Documentation	166
7.22.2.1 PowerManager()	166
7.22.3 Member Function Documentation	167
7.22.3.1 initialize()	167
7.22.3.2 read_device_ids()	167
7.22.3.3 get_current_charge_solar()	167
7.22.3.4 get_current_charge_usb()	167
7.22.3.5 get_current_charge_total()	168
7.22.3.6 get_current_draw()	168
7.22.3.7 get_voltage_battery()	168
7.22.3.8 get_voltage_5v()	168
7.22.3.9 configure()	168
7.22.3.10 is_charging_solar()	168
7.22.3.11 is_charging_usb()	169
7.22.3.12 check_power_alerts()	169
7.22.4 Member Data Documentation	169
7.22.4.1 SOLAR_CURRENT_THRESHOLD	169
7.22.4.2 USB_CURRENT_THRESHOLD	169
7.22.4.3 VOLTAGE_LOW_THRESHOLD	170
7.22.4.4 VOLTAGE_OVERCHARGE_THRESHOLD	170
7.22.4.5 FALL_RATE_THRESHOLD	170
7.22.4.6 FALLING_TREND_REQUIRED	170
7.22.4.7 ina3221_	170
7.22.4.8 initialized_	170
7.22.4.9 powerman_mutex_	170
7.22.4.10 charging_solar_active_	170
7.22.4.11 charging_usb_active_	171
7.23 SensorDataRecord Struct Reference	171
7.23.1 Detailed Description	171
7.23.2 Member Function Documentation	171
7.23.2.1 to_csv()	171
7.23.3 Member Data Documentation	172
7.23.3.1 timestamp	172
7.23.3.2 temperature	172
7.23.3.3 pressure	172

7.23.3.4 humidity	172
7.23.3.5 light	172
7.24 SensorWrapper Class Reference	173
7.24.1 Detailed Description	173
7.24.2 Constructor & Destructor Documentation	174
7.24.2.1 SensorWrapper()	174
7.24.3 Member Function Documentation	174
7.24.3.1 get_instance()	174
7.24.3.2 sensor_init()	175
7.24.3.3 sensor_configure()	175
7.24.3.4 sensor_read_data()	176
7.24.3.5 get_sensor()	177
7.24.3.6 scan_connected_sensors()	177
7.24.3.7 get_available_sensors()	177
7.24.4 Member Data Documentation	178
7.24.4.1 sensors	178
7.25 SystemStateManager Class Reference	178
7.25.1 Detailed Description	179
7.25.2 Constructor & Destructor Documentation	179
7.25.2.1 SystemStateManager() [1/2]	179
7.25.2.2 SystemStateManager() [2/2]	180
7.25.3 Member Function Documentation	181
7.25.3.1 get_instance()	181
7.25.3.2 is_bootloader_reset_pending()	182
7.25.3.3 set_bootloader_reset_pending()	183
7.25.3.4 is_gps_collection_paused()	183
7.25.3.5 set_gps_collection_paused()	183
7.25.3.6 is_sd_card_mounted()	184
7.25.3.7 set_sd_card_mounted()	184
7.25.3.8 get_uart_verbosity()	185
7.25.3.9 set_uart_verbosity()	185
7.25.3.10 operator=()	186
7.25.4 Member Data Documentation	186
7.25.4.1 instance	186
7.25.4.2 instance_mutex	186
7.25.4.3 pending_bootloader_reset	187
7.25.4.4 gps_collection_paused	187
7.25.4.5 sd_card_mounted	187
7.25.4.6 uart_verbosity	187
7.26 TelemetryRecord Struct Reference	187
7.26.1 Detailed Description	188
7.26.2 Member Function Documentation	188

7.26.2.1 to_csv()	188
7.26.3 Member Data Documentation	189
7.26.3.1 timestamp	189
7.26.3.2 build_version	189
7.26.3.3 battery_voltage	189
7.26.3.4 system_voltage	189
7.26.3.5 charge_current_usb	189
7.26.3.6 charge_current_solar	189
7.26.3.7 discharge_current	190
7.26.3.8 time	190
7.26.3.9 latitude	190
7.26.3.10 lat_dir	190
7.26.3.11 longitude	190
7.26.3.12 lon_dir	190
7.26.3.13 speed	191
7.26.3.14 course	191
7.26.3.15 date	191
7.26.3.16 fix_quality	191
7.26.3.17 satellites	191
7.26.3.18 altitude	191
8 File Documentation	193
8.1 build_number.h File Reference	193
8.1.1 Macro Definition Documentation	193
8.1.1.1 BUILD_NUMBER	193
8.2 build_number.h	193
8.3 includes.h File Reference	194
8.4 includes.h	195
8.5 lib/clock/DS3231.cpp File Reference	195
8.6 DS3231.cpp	195
8.7 lib/clock/DS3231.h File Reference	200
8.7.1 Macro Definition Documentation	201
8.7.1.1 DS3231_DEVICE_ADDRESS	201
8.7.1.2 DS3231_SECONDS_REG	201
8.7.1.3 DS3231_MINUTES_REG	202
8.7.1.4 DS3231_HOURS_REG	202
8.7.1.5 DS3231_DAY_REG	202
8.7.1.6 DS3231_DATE_REG	202
8.7.1.7 DS3231_MONTH_REG	202
8.7.1.8 DS3231_YEAR_REG	202
8.7.1.9 DS3231_CONTROL_REG	203
8.7.1.10 DS3231_CONTROL_STATUS_REG	203

8.7.1.11 DS3231_TEMPERATURE_MSB_REG	203
8.7.1.12 DS3231_TEMPERATURE_LSB_REG	203
8.7.2 Enumeration Type Documentation	203
8.7.2.1 days_of_week	203
8.8 DS3231.h	204
8.9 lib/comms/commands/clock_commands.cpp File Reference	205
8.9.1 Macro Definition Documentation	206
8.9.1.1 CLOCK_GROUP	206
8.9.1.2 TIME	206
8.9.1.3 TIMEZONE_OFFSET	206
8.9.1.4 CLOCK_SYNC_INTERVAL	206
8.9.1.5 LAST_SYNC_TIME	206
8.10 clock_commands.cpp	207
8.11 lib/comms/commands/commands.cpp File Reference	209
8.12 commands.cpp	210
8.13 lib/comms/commands/commands.h File Reference	211
8.14 commands.h	213
8.15 lib/comms/commands/diagnostic_commands.cpp File Reference	213
8.16 diagnostic_commands.cpp	214
8.17 lib/comms/commands/event_commands.cpp File Reference	216
8.18 event_commands.cpp	216
8.19 lib/comms/commands/gps_commands.cpp File Reference	217
8.19.1 Macro Definition Documentation	218
8.19.1.1 GPS_GROUP	218
8.19.1.2 POWER_STATUS_COMMAND	218
8.19.1.3 PASSTHROUGH_COMMAND	218
8.19.1.4 RMC_DATA_COMMAND	218
8.19.1.5 GGA_DATA_COMMAND	218
8.20 gps_commands.cpp	219
8.21 lib/comms/commands/power_commands.cpp File Reference	221
8.21.1 Macro Definition Documentation	222
8.21.1.1 POWER_GROUP	222
8.21.1.2 POWER_MANAGER_IDS	222
8.21.1.3 VOLTAGE_BATTERY	222
8.21.1.4 VOLTAGE_MAIN	222
8.21.1.5 CHARGE_USB	223
8.21.1.6 CHARGE_SOLAR	223
8.21.1.7 CHARGE_TOTAL	223
8.21.1.8 DRAW_TOTAL	223
8.22 power_commands.cpp	223
8.23 lib/comms/commands/sensor_commands.cpp File Reference	225
8.23.1 Macro Definition Documentation	226

8.23.1.1 SENSOR_GROUP	226
8.23.1.2 SENSOR_READ	226
8.23.1.3 SENSOR_CONFIGURE	226
8.24 sensor_commands.cpp	227
8.25 lib/comms/commands/storage_commands.cpp File Reference	230
8.25.1 Macro Definition Documentation	231
8.25.1.1 STORAGE_GROUP	231
8.25.1.2 LIST_FILES_COMMAND	231
8.25.1.3 MOUNT_COMMAND	231
8.26 storage_commands.cpp	232
8.27 lib/comms/commands/telemetry_commands.cpp File Reference	233
8.27.1 Macro Definition Documentation	234
8.27.1.1 TELEMETRY_GROUP	234
8.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND	234
8.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND	234
8.27.2 Variable Documentation	234
8.27.2.1 telemetry_mutex	234
8.27.2.2 telemetry_buffer_count	235
8.27.2.3 telemetry_buffer_write_index	235
8.28 telemetry_commands.cpp	235
8.29 lib/comms/communication.cpp File Reference	236
8.29.1 Function Documentation	236
8.29.1.1 initialize_radio()	236
8.29.1.2 lora_tx_done_callback()	237
8.29.2 Variable Documentation	238
8.29.2.1 outgoing	238
8.29.2.2 msgCount	238
8.29.2.3 lastSendTime	238
8.29.2.4 lastReceiveTime	238
8.29.2.5 lastPrintTime	238
8.29.2.6 interval	238
8.30 communication.cpp	239
8.31 lib/comms/communication.h File Reference	239
8.31.1 Function Documentation	240
8.31.1.1 initialize_radio()	240
8.31.1.2 lora_tx_done_callback()	241
8.31.1.3 on_receive()	241
8.31.1.4 handle_uart_input()	242
8.31.1.5 send_message()	243
8.31.1.6 send_frame_uart()	243
8.31.1.7 send_frame_lora()	244
8.31.1.8 split_and_send_message()	244

8.31.1.9 <code>determine_unit()</code>	244
8.32 <code>communication.h</code>	245
8.33 <code>lib/comms/frame.cpp</code> File Reference	245
8.33.1 Detailed Description	246
8.33.2 Typedef Documentation	246
8.33.2.1 <code>CommandHandler</code>	246
8.33.3 Variable Documentation	246
8.33.3.1 <code>eventRegister</code>	246
8.34 <code>frame.cpp</code>	246
8.35 <code>lib/comms/protocol.h</code> File Reference	248
8.35.1 Enumeration Type Documentation	250
8.35.1.1 <code>ErrorCode</code>	250
8.35.1.2 <code>OperationType</code>	251
8.35.1.3 <code>CommandAccessLevel</code>	251
8.35.1.4 <code>ValueUnit</code>	251
8.35.1.5 <code>ExceptionType</code>	252
8.35.1.6 <code>Interface</code>	252
8.35.2 Function Documentation	252
8.35.2.1 <code>exception_type_to_string()</code>	252
8.35.2.2 <code>error_code_to_string()</code>	253
8.35.2.3 <code>operation_type_to_string()</code>	254
8.35.2.4 <code>string_to_operation_type()</code>	255
8.35.2.5 <code>hex_string_to_bytes()</code>	256
8.35.2.6 <code>value_unit_type_to_string()</code>	256
8.35.3 Variable Documentation	258
8.35.3.1 <code>FRAME_BEGIN</code>	258
8.35.3.2 <code>FRAME_END</code>	258
8.35.3.3 <code>DELIMITER</code>	258
8.36 <code>protocol.h</code>	258
8.37 <code>lib/comms/receive.cpp</code> File Reference	259
8.37.1 Detailed Description	260
8.37.2 Macro Definition Documentation	260
8.37.2.1 <code>MAX_PACKET_SIZE</code>	260
8.37.3 Function Documentation	260
8.37.3.1 <code>on_receive()</code>	260
8.37.3.2 <code>handle_uart_input()</code>	261
8.38 <code>receive.cpp</code>	262
8.39 <code>lib/comms/send.cpp</code> File Reference	263
8.39.1 Detailed Description	263
8.39.2 Function Documentation	263
8.39.2.1 <code>send_message()</code>	263
8.39.2.2 <code>send_frame_lora()</code>	264

8.39.2.3 send_frame_uart()	264
8.40 send.cpp	265
8.41 lib/comms/utils_converters.cpp File Reference	265
8.41.1 Detailed Description	266
8.41.2 Function Documentation	266
8.41.2.1 exception_type_to_string()	266
8.41.2.2 value_unit_type_to_string()	266
8.41.2.3 operation_type_to_string()	268
8.41.2.4 string_to_operation_type()	268
8.41.2.5 error_code_to_string()	269
8.41.2.6 hex_string_to_bytes()	270
8.42 utils_converters.cpp	271
8.43 lib/eventman/event_manager.cpp File Reference	272
8.43.1 Detailed Description	273
8.44 event_manager.cpp	273
8.45 lib/eventman/event_manager.h File Reference	274
8.45.1 Detailed Description	276
8.45.2 Macro Definition Documentation	276
8.45.2.1 EVENT_BUFFER_SIZE	276
8.45.2.2 EVENT_FLUSH_THRESHOLD	276
8.45.2.3 EVENT_LOG_FILE	276
8.45.3 Function Documentation	276
8.45.3.1 to_string()	276
8.45.4 Variable Documentation	276
8.45.4.1 id	276
8.45.4.2 timestamp	277
8.45.4.3 group	277
8.45.4.4 event	277
8.46 event_manager.h	277
8.47 lib/location/gps_collector.cpp File Reference	279
8.47.1 Macro Definition Documentation	280
8.47.1.1 MAX_RAW_DATA_LENGTH	280
8.47.2 Function Documentation	280
8.47.2.1 splitString()	280
8.47.2.2 collect_gps_data()	281
8.47.3 Variable Documentation	281
8.47.3.1 nmea_data	281
8.48 gps_collector.cpp	281
8.49 lib/location/gps_collector.h File Reference	282
8.49.1 Function Documentation	283
8.49.1.1 collect_gps_data()	283
8.50 gps_collector.h	284

8.51 lib/location/NMEA/NMEA_data.cpp File Reference	284
8.51.1 Variable Documentation	285
8.51.1.1 nmea_data	285
8.52 NMEA_data.cpp	285
8.53 lib/location/NMEA/NMEA_data.h File Reference	286
8.53.1 Variable Documentation	287
8.53.1.1 nmea_data	287
8.54 NMEA_data.h	287
8.55 lib/pin_config.cpp File Reference	288
8.55.1 Variable Documentation	288
8.55.1.1 lora_cs_pin	288
8.55.1.2 lora_reset_pin	288
8.55.1.3 lora_irq_pin	288
8.55.1.4 lora_address_local	289
8.55.1.5 lora_address_remote	289
8.56 pin_config.cpp	289
8.57 lib/pin_config.h File Reference	289
8.57.1 Macro Definition Documentation	290
8.57.1.1 DEBUG_UART_PORT	290
8.57.1.2 DEBUG_UART_BAUD_RATE	291
8.57.1.3 DEBUG_UART_TX_PIN	291
8.57.1.4 DEBUG_UART_RX_PIN	291
8.57.1.5 MAIN_I2C_PORT	291
8.57.1.6 MAIN_I2C_SDA_PIN	291
8.57.1.7 MAIN_I2C_SCL_PIN	291
8.57.1.8 GPS_UART_PORT	291
8.57.1.9 GPS_UART_BAUD_RATE	291
8.57.1.10 GPS_UART_TX_PIN	292
8.57.1.11 GPS_UART_RX_PIN	292
8.57.1.12 GPS_POWER_ENABLE_PIN	292
8.57.1.13 BUFFER_SIZE	292
8.57.1.14 SD_SPI_PORT	292
8.57.1.15 SD_MISO_PIN	292
8.57.1.16 SD莫斯PIN	292
8.57.1.17 SD_SCK_PIN	292
8.57.1.18 SD_CS_PIN	293
8.57.1.19 SD_CARD_DETECT_PIN	293
8.57.1.20 SX1278_MISO	293
8.57.1.21 SX1278_CS	293
8.57.1.22 SX1278_SCK	293
8.57.1.23 SX1278_MOSI	293
8.57.1.24 SPI_PORT	293

8.57.1.25 READ_BIT	293
8.57.1.26 LORA_DEFAULT_SPI	294
8.57.1.27 LORA_DEFAULT_SPI_FREQUENCY	294
8.57.1.28 LORA_DEFAULT_SS_PIN	294
8.57.1.29 LORA_DEFAULT_RESET_PIN	294
8.57.1.30 LORA_DEFAULT_DIO0_PIN	294
8.57.1.31 PA_OUTPUT_RFO_PIN	294
8.57.1.32 PA_OUTPUT_PA_BOOST_PIN	294
8.57.2 Variable Documentation	295
8.57.2.1 lora_cs_pin	295
8.57.2.2 lora_reset_pin	295
8.57.2.3 lora_irq_pin	295
8.57.2.4 lora_address_local	295
8.57.2.5 lora_address_remote	295
8.58 pin_config.h	296
8.59 lib/powerman/INA3221/INA3221.cpp File Reference	296
8.59.1 Detailed Description	297
8.60 INA3221.cpp	297
8.61 lib/powerman/INA3221/INA3221.h File Reference	301
8.61.1 Detailed Description	303
8.61.2 Enumeration Type Documentation	303
8.61.2.1 ina3221_addr_t	303
8.61.2.2 ina3221_ch_t	303
8.61.2.3 ina3221_reg_t	304
8.61.2.4 ina3221_conv_time_t	304
8.61.2.5 ina3221_avg_mode_t	305
8.61.3 Variable Documentation	305
8.61.3.1 INA3221_CH_NUM	305
8.61.3.2 SHUNT_VOLTAGE_LSB_UV	305
8.62 INA3221.h	306
8.63 lib/powerman/PowerManager.cpp File Reference	308
8.64 PowerManager.cpp	308
8.65 lib/powerman/PowerManager.h File Reference	310
8.66 PowerManager.h	311
8.67 lib/sensors/BH1750/BH1750.cpp File Reference	312
8.68 BH1750.cpp	312
8.69 lib/sensors/BH1750/BH1750.h File Reference	313
8.69.1 Macro Definition Documentation	314
8.69.1.1 _BH1750_DEVICE_ID	314
8.69.1.2 _BH1750_MTREG_MIN	314
8.69.1.3 _BH1750_MTREG_MAX	315
8.69.1.4 _BH1750_DEFAULT_MTREG	315

8.70 BH1750.h	315
8.71 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference	315
8.72 BH1750_WRAPPER.cpp	316
8.73 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference	317
8.74 BH1750_WRAPPER.h	318
8.75 lib/sensors/BME280/BME280.cpp File Reference	318
8.76 BME280.cpp	319
8.77 lib/sensors/BME280/BME280.h File Reference	322
8.78 BME280.h	323
8.79 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference	325
8.80 BME280_WRAPPER.cpp	325
8.81 lib/sensors/BME280/BME280_WRAPPER.h File Reference	326
8.82 BME280_WRAPPER.h	326
8.83 lib/sensors/HMC5883L/HMC5883L.cpp File Reference	327
8.84 HMC5883L.cpp	327
8.85 lib/sensors/HMC5883L/HMC5883L.h File Reference	328
8.86 HMC5883L.h	329
8.87 lib/sensors/HMC5883L/HMC5883L_WRAPPER.cpp File Reference	329
8.88 HMC5883L_WRAPPER.cpp	330
8.89 lib/sensors/HMC5883L/HMC5883L_WRAPPER.h File Reference	331
8.90 HMC5883L_WRAPPER.h	332
8.91 lib/sensors/ISensor.cpp File Reference	332
8.91.1 Detailed Description	332
8.92 ISensor.cpp	333
8.93 lib/sensors/ISensor.h File Reference	334
8.93.1 Enumeration Type Documentation	335
8.93.1.1 SensorType	335
8.93.1.2 SensorDataTypeldentifier	335
8.94 ISensor.h	336
8.95 lib/sensors/MPU6050/MPU6050.cpp File Reference	336
8.96 MPU6050.cpp	336
8.97 lib/sensors/MPU6050/MPU6050.h File Reference	337
8.98 MPU6050.h	337
8.99 lib/sensors/MPU6050/MPU6050_WRAPPER.cpp File Reference	337
8.100 MPU6050_WRAPPER.cpp	337
8.101 lib/sensors/MPU6050/MPU6050_WRAPPER.h File Reference	337
8.102 MPU6050_WRAPPER.h	338
8.103 lib/storage/storage.cpp File Reference	339
8.103.1 Detailed Description	339
8.103.2 Function Documentation	340
8.103.2.1 fs_init()	340
8.103.2.2 fs_stop()	341

8.104 storage.cpp	341
8.105 lib/storage/storage.h File Reference	342
8.105.1 Function Documentation	343
8.105.1.1 fs_init()	343
8.105.1.2 fs_open_file()	344
8.105.1.3 fs_write_file()	344
8.105.1.4 fs_read_file()	344
8.105.1.5 fs_close_file()	344
8.105.1.6 fs_file_exists()	344
8.105.2 Variable Documentation	344
8.105.2.1 sd_card_mounted	344
8.106 storage.h	345
8.107 lib/system_state_manager.cpp File Reference	345
8.107.1 Detailed Description	346
8.108 system_state_manager.cpp	346
8.109 lib/system_state_manager.h File Reference	347
8.109.1 Detailed Description	348
8.110 system_state_manager.h	348
8.111 lib/telemetry/telemetry_manager.cpp File Reference	349
8.111.1 Detailed Description	350
8.111.2 Macro Definition Documentation	350
8.111.2.1 TELEMETRY_CSV_PATH	350
8.111.2.2 SENSOR_DATA_CSV_PATH	351
8.111.2.3 DEFAULT_SAMPLE_INTERVAL_MS	351
8.111.2.4 DEFAULT_FLUSH_THRESHOLD	351
8.111.3 Variable Documentation	351
8.111.3.1 powerManager	351
8.111.3.2 systemClock	351
8.111.3.3 nmea_data	351
8.111.3.4 sample_interval_ms	351
8.111.3.5 flush_threshold	352
8.112 telemetry_manager.cpp	352
8.113 lib/telemetry/telemetry_manager.h File Reference	356
8.113.1 Detailed Description	357
8.114 telemetry_manager.h	358
8.115 lib/utils.cpp File Reference	359
8.115.1 Detailed Description	360
8.115.2 Function Documentation	360
8.115.2.1 get_level_color()	360
8.115.2.2 get_level_prefix()	362
8.115.2.3 uart_print()	363
8.115.3 Variable Documentation	365

8.115.3.1 uart_mutex	365
8.116 utils.cpp	366
8.117 lib/utils.h File Reference	366
8.117.1 Detailed Description	367
8.117.2 Macro Definition Documentation	368
8.117.2.1 ANSI_RED	368
8.117.2.2 ANSI_GREEN	368
8.117.2.3 ANSI_YELLOW	368
8.117.2.4 ANSI_BLUE	368
8.117.2.5 ANSI_RESET	368
8.117.3 Enumeration Type Documentation	368
8.117.3.1 VerbosityLevel	368
8.117.4 Function Documentation	369
8.117.4.1 uart_print()	369
8.118 utils.h	370
8.119 main.cpp File Reference	371
8.119.1 Macro Definition Documentation	372
8.119.1.1 LOG_FILENAME	372
8.119.2 Function Documentation	372
8.119.2.1 core1_entry()	372
8.119.2.2 init_pico_hw()	373
8.119.2.3 init_modules()	373
8.119.2.4 main()	374
8.119.3 Variable Documentation	374
8.119.3.1 powerManager	374
8.119.3.2 systemClock	375
8.119.3.3 buffer	375
8.119.3.4 buffer_index	375
8.120 main.cpp	375
8.121 test/comms/commands/test_clock_commands.cpp File Reference	377
8.122 test_clock_commands.cpp	377
8.123 test/comms/commands/test_diagnostic_commands.cpp File Reference	377
8.123.1 Function Documentation	378
8.123.1.1 test_handle_get_commands_list()	378
8.123.1.2 test_handle_get_build_version()	379
8.123.1.3 test_handle_verbosity()	379
8.123.1.4 test_handle_enter_bootloader_mode()	380
8.124 test_diagnostic_commands.cpp	380
8.125 test/comms/commands/test_event_commands.cpp File Reference	381
8.126 test_event_commands.cpp	381
8.127 test/comms/commands/test_gps_commands.cpp File Reference	381
8.128 test_gps_commands.cpp	381

8.129 test/comms/commands/test_power_commands.cpp File Reference	381
8.130 test_power_commands.cpp	381
8.131 test/comms/commands/test_sensor_commands.cpp File Reference	382
8.132 test_sensor_commands.cpp	382
8.133 test/comms/commands/test_storage_commands.cpp File Reference	382
8.134 test_storage_commands.cpp	382
8.135 test/comms/commands/test_telemetry_commands.cpp File Reference	382
8.136 test_telemetry_commands.cpp	382
8.137 test/comms/test_command_handlers.cpp File Reference	382
8.137.1 Function Documentation	383
8.137.1.1 send_frame_uart()	383
8.137.1.2 send_frame_lora()	384
8.137.1.3 setUp()	384
8.137.1.4 tearDown()	384
8.137.1.5 test_command_handler_get_operation()	385
8.137.1.6 test_command_handler_set_operation()	385
8.137.1.7 test_command_handler_invalid_operation()	386
8.137.2 Variable Documentation	386
8.137.2.1 uart_send_called	386
8.137.2.2 lora_send_called	387
8.137.2.3 last_frame_sent	387
8.138 test_comand_handlers.cpp	387
8.139 test/comms/test_converters.cpp File Reference	388
8.139.1 Function Documentation	388
8.139.1.1 test_operation_type_conversion()	388
8.139.1.2 test_value_unit_type_conversion()	389
8.139.1.3 test_exception_type_conversion()	389
8.139.1.4 test_hex_string_conversion()	390
8.140 test_converters.cpp	390
8.141 test/comms/test_frame_build.cpp File Reference	391
8.141.1 Function Documentation	391
8.141.1.1 test_frame_build_val()	391
8.141.1.2 test_frame_build_err()	392
8.141.1.3 test_frame_build_get()	392
8.141.1.4 test_frame_build_set()	393
8.141.1.5 test_frame_build_res()	393
8.141.1.6 test_frame_build_seq()	394
8.142 test_frame_build.cpp	395
8.143 test/comms/test_frame_coding.cpp File Reference	396
8.143.1 Function Documentation	396
8.143.1.1 test_frame_encode_basic()	396
8.143.1.2 test_frame_decode_basic()	397

8.143.1.3 test_frame_decode_invalid_header()	397
8.144 test_frame_coding.cpp	398
8.145 test/comms/test_frame_common.h File Reference	398
8.145.1 Function Documentation	399
8.145.1.1 create_test_frame()	399
8.146 test_frame_common.h	400
8.147 test/comms/test_frame_send.cpp File Reference	400
8.147.1 Function Documentation	400
8.147.1.1 setUp()	400
8.147.1.2 tearDown()	401
8.147.1.3 test_send_frame_uart()	401
8.148 test_frame_send.cpp	401
8.149 test/mocks/hardwareMocks.cpp File Reference	402
8.149.1 Function Documentation	402
8.149.1.1 mock_uart_puts()	402
8.149.1.2 mock_uart_init()	403
8.149.1.3 mock_spi_write_blocking()	403
8.149.1.4 mock_spi_read_blocking()	403
8.149.2 Variable Documentation	403
8.149.2.1 mock_uart_enabled	403
8.149.2.2 uart_output_buffer	403
8.149.2.3 mock_spi_enabled	403
8.149.2.4 spi_output_buffer	404
8.150 hardwareMocks.cpp	404
8.151 test/mocks/hardwareMocks.h File Reference	404
8.151.1 Function Documentation	406
8.151.1.1 mock_uart_puts()	406
8.151.1.2 mock_uart_init()	406
8.151.1.3 mock_spi_write_blocking()	406
8.151.1.4 mock_spi_read_blocking()	406
8.151.2 Variable Documentation	406
8.151.2.1 mock_uart_enabled	406
8.151.2.2 uart_output_buffer	406
8.151.2.3 mock_spi_enabled	407
8.151.2.4 spi_output_buffer	407
8.152 hardwareMocks.h	407
8.153 test/testMocks.cpp File Reference	407
8.153.1 Function Documentation	408
8.153.1.1 uart_print()	408
8.153.1.2 test_error_code_conversion()	408
8.153.1.3 test_command_handler_get_operation()	409
8.153.1.4 test_command_handler_set_operation()	409

8.153.1.5 test_command_handler_invalid_operation()	409
8.153.2 Variable Documentation	409
8.153.2.1 powerManager	409
8.153.2.2 systemClock	409
8.153.2.3 nmea_data	409
8.153.2.4 g_pending_bootloader_reset	409
8.153.2.5 g_uart_verbosity	410
8.153.2.6 uart_output_buffer	410
8.153.2.7 mock_uart_enabled	410
8.154 test_mocks.cpp	410
8.155 test/test_runner.cpp File Reference	411
8.155.1 Function Documentation	412
8.155.1.1 test_frame_encode_basic()	412
8.155.1.2 test_frame_decode_basic()	413
8.155.1.3 test_frame_decode_invalid_header()	413
8.155.1.4 test_frame_build_get()	414
8.155.1.5 test_frame_build_set()	414
8.155.1.6 test_frame_build_res()	415
8.155.1.7 test_frame_build_seq()	416
8.155.1.8 test_frame_build_val()	416
8.155.1.9 test_frame_build_err()	417
8.155.1.10 test_operation_type_conversion()	417
8.155.1.11 test_value_unit_type_conversion()	418
8.155.1.12 test_exception_type_conversion()	418
8.155.1.13 test_hex_string_conversion()	419
8.155.1.14 test_command_handler_get_operation()	419
8.155.1.15 test_command_handler_set_operation()	420
8.155.1.16 test_command_handler_invalid_operation()	421
8.155.1.17 test_handle_get_commands_list()	421
8.155.1.18 test_handle_get_build_version()	422
8.155.1.19 test_handle_verbosity()	423
8.155.1.20 test_handle_enter_bootloader_mode()	423
8.155.1.21 test_error_code_conversion()	424
8.155.1.22 main()	424
8.156 test_runner.cpp	425
Index	427

Chapter 1

Clock Commands

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

Command ID: 3.3

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

Command ID: 2.5

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

Command ID: 2.6

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

Command ID: 2.4

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

Command ID: 2.7

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

Command ID: 5.2

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

Command ID: 7.4

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

Command ID: 5.1

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

Command ID: 3.7

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

Command ID: 8.2

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

Command ID: 2.0

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

Command ID: 7.3

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

Command ID: 3.0

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

Command ID: 4.2

Member `handle_get_voltage_5v (const std::string ¶m, OperationType operationType)`

Command ID: 2.3

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

Command ID: 2.2

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

Command ID: 7.1

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

Command ID: 6.0

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

Command ID: 6.4

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

Command ID: 3.1

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 2

Topic Index

2.1 Topics

Here is a list of all topics with brief descriptions:

Clock Management Commands	13
Command System	17
Diagnostic Commands	19
Event Commands	25
GPS Commands	27
Power Commands	31
Sensor Commands	37
Storage Commands	40
Telemetry Buffer Commands	42
Frame Handling	44
Event Manager	49
INA3221 Power Monitor	56
Configuration Functions	56
Measurement Functions	66
Alert Functions	68
Telemetry Manager	74

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

BH1750	85
BME280	91
BME280CalibParam	100
INA3221::conf_reg_t	106
DS3231	108
ds3231_data_t	126
EventEmitter	128
EventLog	130
EventManager	132
EventManagerImpl	137
FileHandle	140
Frame	141
HMC5883L	143
INA3221	149
ISensor	155
BH1750Wrapper	88
BME280Wrapper	104
HMC5883LWrapper	146
MPU6050Wrapper	160
INA3221::masken_reg_t	157
NMEAData	162
PowerManager	165
SensorDataRecord	171
SensorWrapper	173
SystemStateManager	178
TelemetryRecord	187

Chapter 4

Class Index

4.1 Class List

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

BH1750	85
BH1750Wrapper	88
BME280	91
BME280CalibParam	100
BME280Wrapper	104
INA3221::conf_reg_t Configuration register bit fields	106
DS3231	108
Class for interfacing with the DS3231 real-time clock	108
ds3231_data_t	126
Structure to hold time and date information from DS3231	126
EventEmitter	128
Provides a static method for emitting events	128
EventLog	130
Represents a single event log entry	130
EventManager	132
Manages the event logging system	132
EventManagerImpl	137
Implementation of the EventManager class	137
FileHandle	140
Frame	141
Represents a communication frame used for data exchange	141
HMC5883L	143
HMC5883LWrapper	146
INA3221	149
INA3221 Triple-Channel Power Monitor driver class	149
ISensor	155
INA3221::masken_reg_t	157
Mask/Enable register bit fields	157
MPU6050Wrapper	160
NMEAData	162
PowerManager	165
SensorDataRecord	171
SensorWrapper	173
Manages different sensor types and provides a unified interface for accessing sensor data	173

SystemStateManager	
Singleton class for managing global system states	178
TelemetryRecord	
Structure representing a single telemetry data point	187

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

build_number.h	193
includes.h	194
main.cpp	371
lib/pin_config.cpp	288
lib/pin_config.h	289
lib/system_state_manager.cpp Implementation of the SystemStateManager singleton class	345
lib/system_state_manager.h Declaration of the SystemStateManager singleton class for managing global system states	347
lib/utils.cpp Implementation of utility functions for the Kabisat firmware	359
lib/utils.h Utility functions and definitions for the Kabisat firmware	366
lib/clock/DS3231.cpp	195
lib/clock/DS3231.h	200
lib/comms/communication.cpp	236
lib/comms/communication.h	239
lib/comms/frame.cpp Implements functions for encoding, decoding, building, and processing Frames	245
lib/comms/protocol.h	248
lib/comms/receive.cpp Implements functions for receiving and processing data, including LoRa and UART input	259
lib/comms/send.cpp Implements functions for sending data, including LoRa messages and Frames	263
lib/comms/utils_converters.cpp Implements utility functions for converting between different data types	265
lib/comms/commands/clock_commands.cpp	205
lib/comms/commands/commands.cpp	209
lib/comms/commands/commands.h	211
lib/comms/commands/diagnostic_commands.cpp	213
lib/comms/commands/event_commands.cpp	216
lib/comms/commands/gps_commands.cpp	217
lib/comms/commands/power_commands.cpp	221
lib/comms/commands/sensor_commands.cpp	225
lib/comms/commands/storage_commands.cpp	230

lib/comms/commands/ telemetry_commands.cpp	233
lib/eventman/ event_manager.cpp	
Implements the event management system for the Kubisat firmware	272
lib/eventman/ event_manager.h	
Manages the event logging system for the Kubisat firmware	274
lib/location/ gps_collector.cpp	279
lib/location/ gps_collector.h	282
lib/location/NMEA/ NMEA_data.cpp	284
lib/location/NMEA/ NMEA_data.h	286
lib/powerman/ PowerManager.cpp	308
lib/powerman/ PowerManager.h	310
lib/powerman/INA3221/ INA3221.cpp	
Implementation of the INA3221 power monitor driver	296
lib/powerman/INA3221/ INA3221.h	
Header file for the INA3221 triple-channel power monitor driver	301
lib/sensors/ ISensor.cpp	
Implements the SensorWrapper class for managing different sensor types	332
lib/sensors/ ISensor.h	334
lib/sensors/BH1750/ BH1750.cpp	312
lib/sensors/BH1750/ BH1750.h	313
lib/sensors/BH1750/ BH1750_WRAPPER.cpp	315
lib/sensors/BH1750/ BH1750_WRAPPER.h	317
lib/sensors/BME280/ BME280.cpp	318
lib/sensors/BME280/ BME280.h	322
lib/sensors/BME280/ BME280_WRAPPER.cpp	325
lib/sensors/BME280/ BME280_WRAPPER.h	326
lib/sensors/HMC5883L/ HMC5883L.cpp	327
lib/sensors/HMC5883L/ HMC5883L.h	328
lib/sensors/HMC5883L/ HMC5883L_WRAPPER.cpp	329
lib/sensors/HMC5883L/ HMC5883L_WRAPPER.h	331
lib/sensors/MPU6050/ MPU6050.cpp	336
lib/sensors/MPU6050/ MPU6050.h	337
lib/sensors/MPU6050/ MPU6050_WRAPPER.cpp	337
lib/sensors/MPU6050/ MPU6050_WRAPPER.h	337
lib/storage/ storage.cpp	
Implements file system operations for the Kubisat firmware	339
lib/storage/ storage.h	342
lib/telemetry/ telemetry_manager.cpp	
Implementation of telemetry collection and storage functionality	349
lib/telemetry/ telemetry_manager.h	
System telemetry collection and logging	356
test/ testMocks.cpp	407
test/ testRunner.cpp	411
test/comms/ testCommandHandlers.cpp	382
test/comms/ testConverters.cpp	388
test/comms/ testFrameBuild.cpp	391
test/comms/ testFrameCoding.cpp	396
test/comms/ testFrameCommon.h	398
test/comms/ testFrameSend.cpp	400
test/comms/commands/ testClockCommands.cpp	377
test/comms/commands/ testDiagnosticCommands.cpp	377
test/comms/commands/ testEventCommands.cpp	381
test/comms/commands/ testGpsCommands.cpp	381
test/comms/commands/ testPowerCommands.cpp	381
test/comms/commands/ testSensorCommands.cpp	382
test/comms/commands/ testStorageCommands.cpp	382
test/comms/commands/ testTelemetryCommands.cpp	382
test/mocks/ hardwareMocks.cpp	402

test/mocks/hardware_mock.h	404
----------------------------	-------	-----

Chapter 6

Topic Documentation

6.1 Clock Management Commands

Commands for managing system time and clock settings.

Functions

- std::vector< Frame > handle_time (const std::string ¶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_clock_sync_interval (const std::string ¶m, OperationType operationType)
Handler for getting and setting clock synchronization interval.
- std::vector< Frame > handle_get_last_sync_time (const std::string ¶m, OperationType operationType)
Handler for getting last clock sync time.

Variables

- DS3231 systemClock

6.1.1 Detailed Description

Commands for managing system time and clock settings.

6.1.2 Function Documentation

6.1.2.1 handle_time()

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

Handler for getting and setting system time.

Parameters

<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**

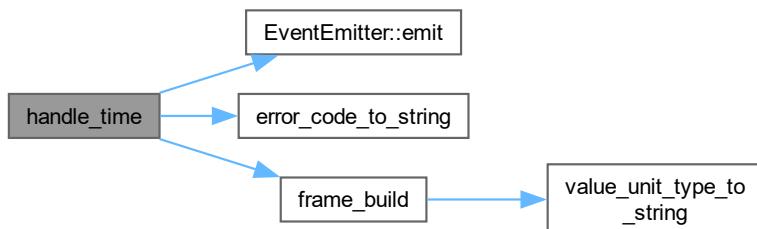
When setting time, expects Unix timestamp as parameter

Command

Command ID: 3.0

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

Here is the call graph for this function:



6.1.2.2 handle_timezone_offset()

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

Handler for getting and setting timezone offset.

Parameters

<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

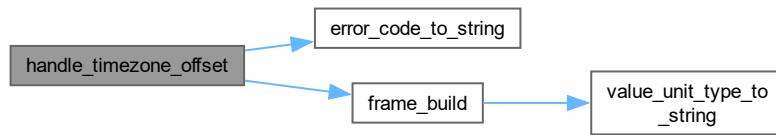
GET: **KBST;0;GET;3;1;;KBST**

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

Command Command ID: 3.1

Definition at line 97 of file [clock_commands.cpp](#).

Here is the call graph for this function:

**6.1.2.3 handle_clock_sync_interval()**

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

Handler for getting and setting clock synchronization interval.

Parameters

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

Returns

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

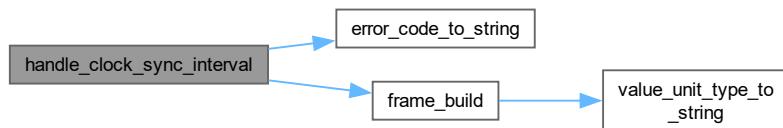
Note

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

Command Command ID: 3.3

Definition at line 156 of file [clock_commands.cpp](#).

Here is the call graph for this function:



6.1.2.4 handle_get_last_sync_time()

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

Handler for getting last clock sync time.

Parameters

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

Returns

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

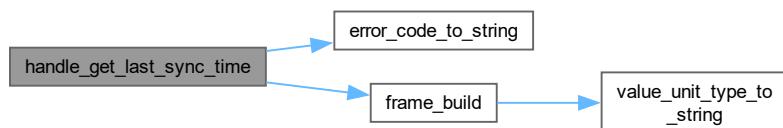
Note

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

Command Command ID: 3.7

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

Here is the call graph for this function:



6.1.3 Variable Documentation

6.1.3.1 systemClock

```
DS3231 systemClock [extern]
```

6.2 Command System

Core command system implementation.

Typedefs

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

Functions

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

Variables

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

6.2.1 Detailed Description

Core command system implementation.

6.2.2 Typedef Documentation

6.2.2.1 CommandHandler

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

Function type for command handlers.

Definition at line 15 of file `commands.cpp`.

6.2.2.2 CommandMap

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

Map type for storing command handlers.

Definition at line 21 of file [commands.cpp](#).

6.2.3 Function Documentation

6.2.3.1 execute_command()

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

Executes a command based on its key.

Parameters

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

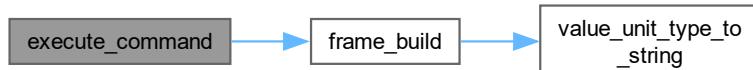
Returns

[Frame](#) Response frame containing execution result

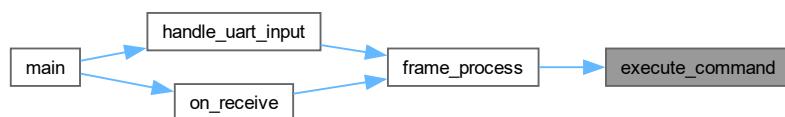
Looks up the command handler in commandHandlers map and executes it

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.4 Variable Documentation

6.2.4.1 command_handlers

`CommandMap command_handlers`

Initial value:

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

Global map of all command handlers.

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

Definition at line 27 of file `commands.cpp`.

6.3 Diagnostic Commands

Functions

- `std::vector< Frame > handle_get_commands_list (const std::string ¶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_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.

6.3.1 Detailed Description

6.3.2 Function Documentation

6.3.2.1 handle_get_commands_list()

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

Handler for listing all available commands on UART.

Parameters

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

Returns

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

Note

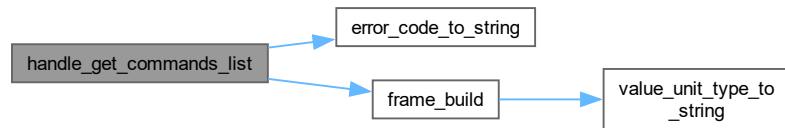
KBST;0;GET;1;0;;TSBK

Print all available commands on UART port

Command Command ID: 0

Definition at line 21 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.2.2 handle_get_build_version()

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

Get firmware build version.

Parameters

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

Returns

One-element vector with result frame

Note

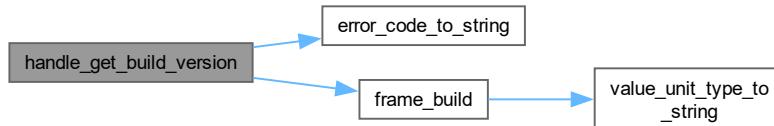
KBST;0;GET;1;1;;TSBK

Get the firmware build version

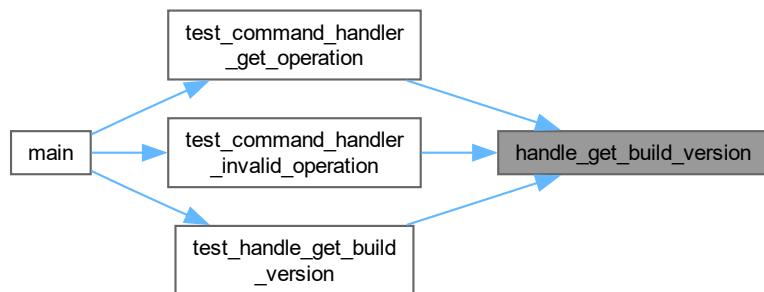
Command Command ID: 1

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.2.3 handle_verbosity()

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

Handles setting or getting the UART verbosity level.

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

Parameters

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

Returns

Vector containing one frame indicating the result of the operation.

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

Note

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

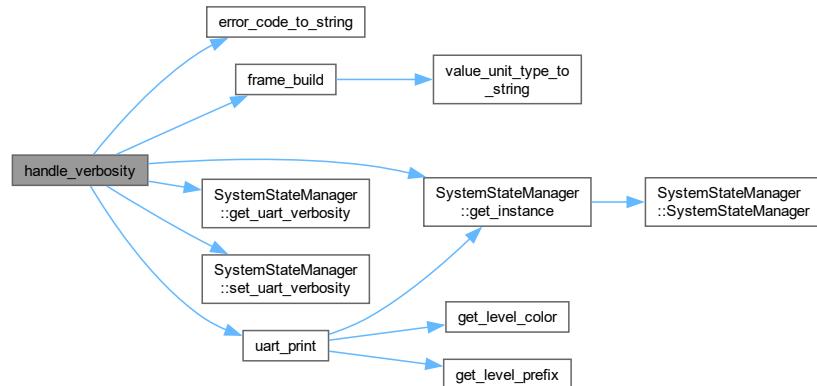
KBST;0;SET;1;8;[level];TSBK - Sets the verbosity level.

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

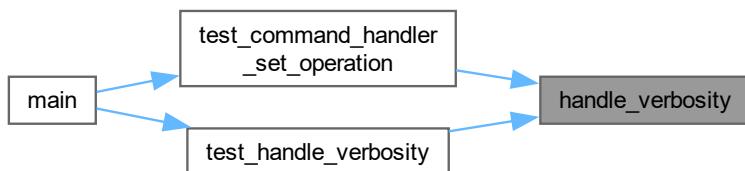
Command Command ID: 1.8

Definition at line 117 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.2.4 handle_enter_bootloader_mode()

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

Reboot system to USB firmware loader.

Parameters

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

Returns

Frame with operation result

Note

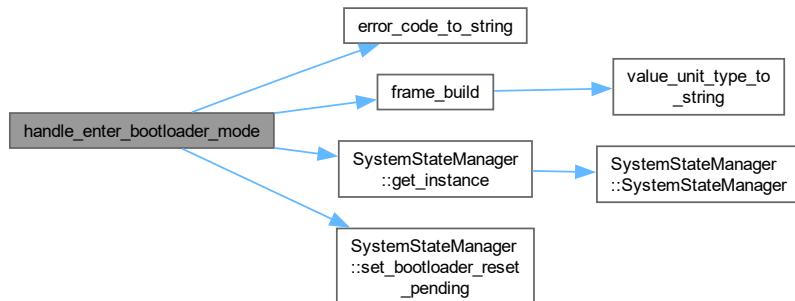
KBST;0;SET;1;9;;TSBK

Reboot the system to USB firmware loader

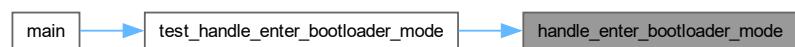
Command Command ID: 2

Definition at line 157 of file [diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.4 Event Commands

Commands for accessing and managing system event logs.

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.

6.4.1 Detailed Description

Commands for accessing and managing system event logs.

6.4.2 Function Documentation

6.4.2.1 [handle_get_last_events\(\)](#)

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

Handler for retrieving last N events from the event log.

Parameters

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

Returns

[Frame](#) containing:

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

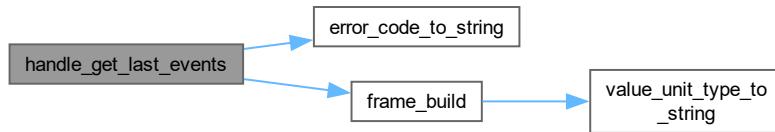
Note

KBST;0;GET;5;1;[N];TSBK - Retrieves the last N events. If N is 0, retrieves all events.
Returns up to 10 most recent events per frame.

Command Command ID: 5.1

Definition at line 33 of file [event_commands.cpp](#).

Here is the call graph for this function:

**6.4.2.2 handle_get_event_count()**

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

Handler for getting total number of events in the log.

Parameters

<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

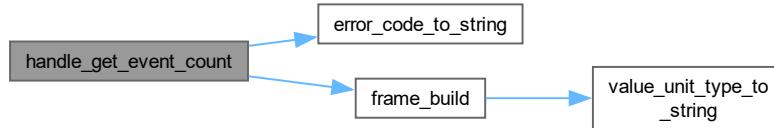
KBST;0;GET;5;2;;TSBK

Returns the total number of events in the log

Command Command ID: 5.2

Definition at line 100 of file [event_commands.cpp](#).

Here is the call graph for this function:



6.5 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)
- `std::vector< Frame > handle_get_rmc_data (const std::string ¶m, OperationType operationType)`
Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.
- `std::vector< Frame > handle_get_gga_data (const std::string ¶m, OperationType operationType)`
Handler for retrieving GPS GGA (Global Positioning System Fix Data) data.

6.5.1 Detailed Description

Commands for controlling and monitoring the GPS module.

6.5.2 Function Documentation

6.5.2.1 handle_gps_power_status()

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

Handler for controlling GPS module power state.

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

Return current GPS module power state: ON/OFF

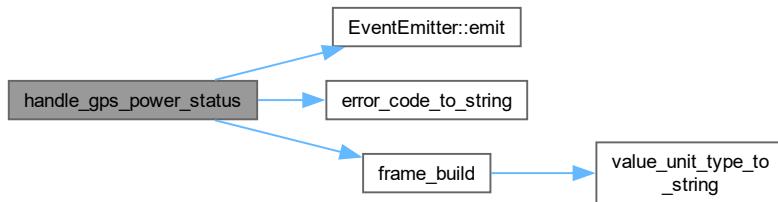
KBST;0;SET;7;1;POWER;TSBK

POWER - 0 - OFF, 1 - ON

Command Command ID: 7.1

Definition at line 33 of file [gps_commands.cpp](#).

Here is the call graph for this function:

**6.5.2.2 handle_enable_gps_uart_passthrough()**

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

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

Parameters

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

Returns

Vector of Frames containing:

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

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

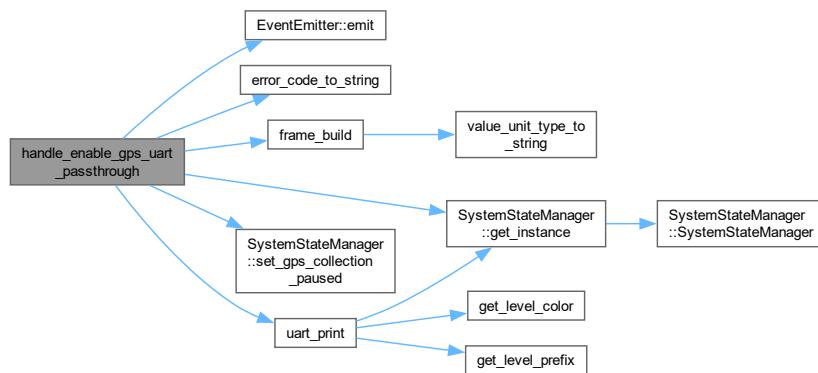
TIMEOUT - 1-600s, default 60s

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 90 of file [gps_commands.cpp](#).

Here is the call graph for this function:

**6.5.2.3 handle_get_rmc_data()**

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

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

Parameters

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

Returns

Vector of Frames containing:

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

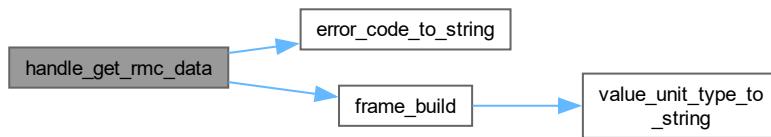
Note

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

Command Command ID: 7.3

Definition at line 193 of file [gps_commands.cpp](#).

Here is the call graph for this function:



6.5.2.4 handle_get_gga_data()

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

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

Parameters

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

Returns

Vector of Frames containing:

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

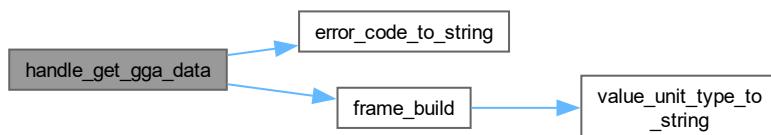
Note

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

Command Command ID: 7.4

Definition at line 235 of file [gps_commands.cpp](#).

Here is the call graph for this function:



6.6 Power Commands

Commands for monitoring power subsystem and battery management.

Functions

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

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

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

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

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

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

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

Handler for getting system current draw.

6.6.1 Detailed Description

Commands for monitoring power subsystem and battery management.

6.6.2 Function Documentation

6.6.2.1 handle_get_power_manager_ids()

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

Handler for retrieving Power Manager IDs.

Parameters

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

Returns

Vector of Frames containing:

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

Note

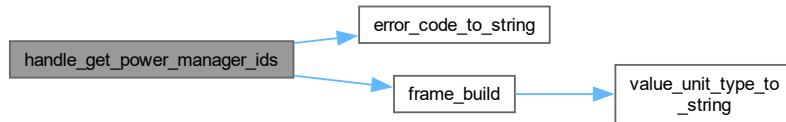
KBST;0;GET;2;0;;TSBK

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

Command Command ID: 2.0

Definition at line 30 of file [power_commands.cpp](#).

Here is the call graph for this function:

**6.6.2.2 handle_get_voltage_battery()**

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

Handler for getting battery voltage.

Parameters

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

Returns

Vector of Frames containing:

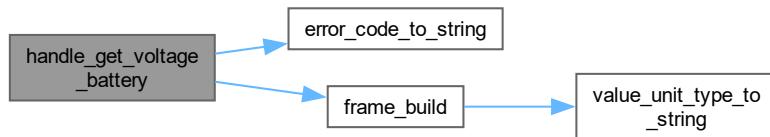
- Success: Battery voltage in Volts
- Error: Error message

Note**KBST;0;GET;2;2;;TSBK**

This command is used to retrieve the battery voltage

Command Command ID: 2.2Definition at line 64 of file [power_commands.cpp](#).

Here is the call graph for this function:

**6.6.2.3 handle_get_voltage_5v()**

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

Handler for getting 5V rail voltage.

Parameters

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

Returns

Vector of Frames containing:

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

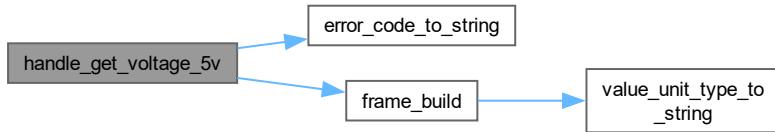
Note**KBST;0;GET;2;3;;TSBK**

This command is used to retrieve the 5V rail voltage

Command Command ID: 2.3

Definition at line 98 of file [power_commands.cpp](#).

Here is the call graph for this function:



6.6.2.4 handle_get_current_charge_usb()

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

Handler for getting USB charge current.

Parameters

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

Returns

Vector of Frames containing:

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

Note

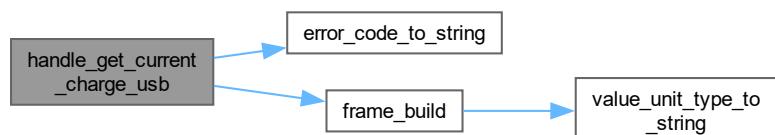
KBST;0;GET;2;4;;TSBK

This command is used to retrieve the USB charge current

Command Command ID: 2.4

Definition at line 132 of file [power_commands.cpp](#).

Here is the call graph for this function:



6.6.2.5 handle_get_current_charge_solar()

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

Handler for getting solar panel charge current.

Parameters

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

Returns

Vector of Frames containing:

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

Note

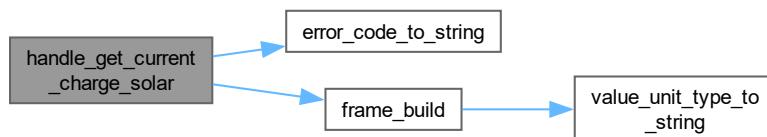
KBST;0;GET;2;5;;TSBK

This command is used to retrieve the solar panel charge current

Command Command ID: 2.5

Definition at line 166 of file [power_commands.cpp](#).

Here is the call graph for this function:



6.6.2.6 handle_get_current_charge_total()

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

Handler for getting total charge current.

Parameters

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

Returns

Vector of Frames containing:

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

Note

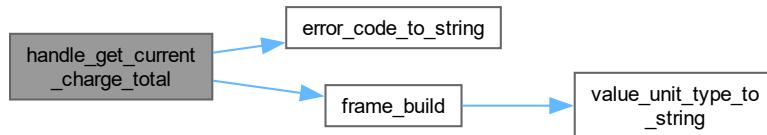
KBST;0;GET;2;6;;TSBK

This command is used to retrieve the total charge current

Command Command ID: 2.6

Definition at line 200 of file [power_commands.cpp](#).

Here is the call graph for this function:



6.6.2.7 handle_get_current_draw()

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

Handler for getting system current draw.

Parameters

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

Returns

Vector of Frames containing:

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

Note

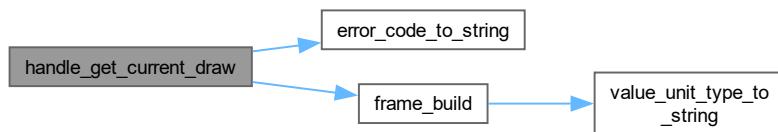
KBST;0;GET;2;7;;TSBK

This command is used to retrieve the system current draw

Command Command ID: 2.7

Definition at line 234 of file [power_commands.cpp](#).

Here is the call graph for this function:



6.7 Sensor Commands

Commands for reading and configuring sensors.

Functions

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

6.7.1 Detailed Description

Commands for reading and configuring sensors.

6.7.2 Function Documentation

6.7.2.1 handle_get_sensor_data()

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

Handler for reading sensor data.

Parameters

<i>param</i>	String in format "sensor_type[-data_type]" where: <ul style="list-style-type: none"> • sensor_type: "light", "environment", "magnetometer", "imu" • data_type (optional): specific data type for the sensor <ul style="list-style-type: none"> – For light: "light_level" – For environment: "temperature", "pressure", "humidity" – For magnetometer: "mag_field_x", "mag_field_y", "mag_field_z" – For IMU: "gyro_x", "gyro_y", "gyro_z", "accel_x", "accel_y", "accel_z"
<i>operationType</i>	GET

Returns

Vector of Frames containing:

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

Note

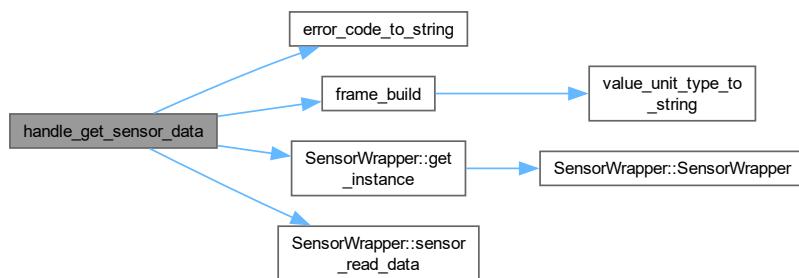
KBST;0;GET;3;0;light-light_level;TSBK

This command is used to read data from sensors

Command Command ID: 3.0

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

Here is the call graph for this function:



6.7.2.2 handle_sensor_config()

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

Handler for configuring sensors.

Parameters

<i>param</i>	String in format "sensor_type;key1:value1 key2:value2 ..." <ul style="list-style-type: none">• sensor_type: "light", "environment", "magnetometer", "imu"• key-value pairs for configuration parameters
<i>operationType</i>	SET

Returns

Vector of Frames containing:

- Success: Success message
- Error: Error message

Note

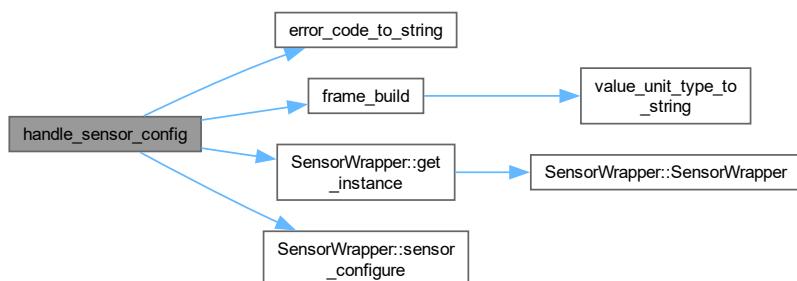
KBST;0;SET;3;1;light;measurement_mode:continuously_high_resolution;TSBK

This command is used to configure sensors

Command Command ID: 3.1

Definition at line 236 of file [sensor_commands.cpp](#).

Here is the call graph for this function:

6.7.2.3 `handle_get_sensor_list()`

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

Handler for listing available sensors.

Parameters

<i>param</i>	Empty string or optional filter criteria
<i>operationType</i>	GET

Returns

Vector of Frames containing:

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

Note

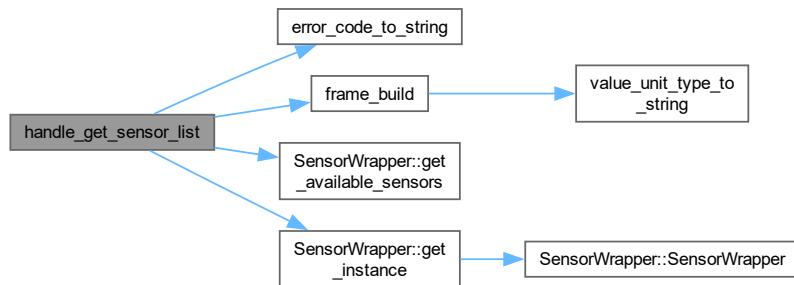
KBST;0;GET;4;2;;TSBK (lists all sensors)

This command is used to get a list of available sensors

Command Command ID: 4.2

Definition at line 320 of file [sensor_commands.cpp](#).

Here is the call graph for this function:



6.8 Storage Commands

Commands for interacting with the SD card storage.

Functions

- `std::vector< Frame > handle_list_files (const std::string ¶m, OperationType operationType)`
Handles the list files command.
- `std::vector< Frame > handle_mount (const std::string ¶m, OperationType operationType)`
Handles the SD card mount/unmount command.

6.8.1 Detailed Description

Commands for interacting with the SD card storage.

6.8.2 Function Documentation

6.8.2.1 handle_list_files()

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

Handles the list files command.

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

Parameters

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

Note

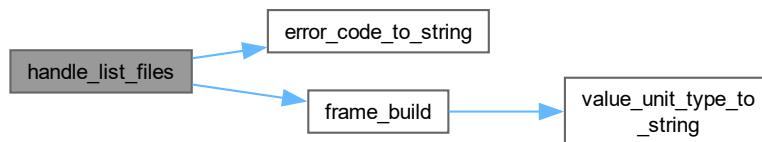
KBST;0;GET;6;0;;TSBK

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

Command Command ID: 6.0

Definition at line 37 of file [storage_commands.cpp](#).

Here is the call graph for this function:



6.8.2.2 handle_mount()

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

Handles the SD card mount/unmount command.

This function mounts or unmounts the SD card.

Parameters

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

Returns

A vector of Frames indicating the result of the operation.

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

Note

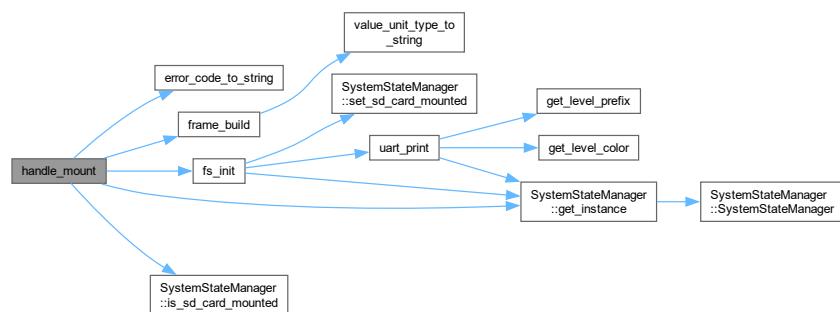
KBST;0;SET;6;4;[0|1];TSBK

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

Command Command ID: 6.4

Definition at line 123 of file [storage_commands.cpp](#).

Here is the call graph for this function:



6.9 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.

6.9.1 Detailed Description

Commands for interacting with the telemetry buffer.

6.9.2 Function Documentation

6.9.2.1 handle_get_last_telemetry_record()

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

Handles the get last record command.

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

Parameters

<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

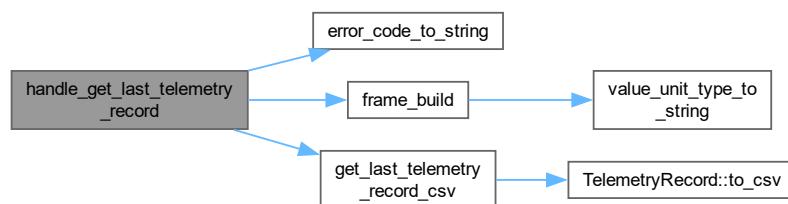
This command retrieves the last telemetry record from the buffer and sends it base64 encoded.

Command

Command ID: 8.2

Definition at line 37 of file `telemetry_commands.cpp`.

Here is the call graph for this function:



6.9.2.2 handle_get_last_sensor_record()

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

Handles the get last sensor record command.

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

Parameters

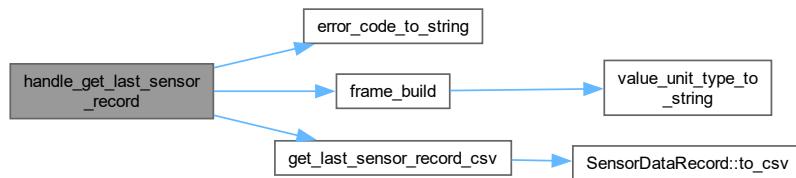
<i>param</i>	Unused.
<i>operationType</i>	The operation type (must be GET).

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:



6.10 Frame Handling

Functions for encoding, decoding and building communication frames.

Functions

- `std::string frame_encode (const Frame &frame)`
Encodes a `Frame` instance into a string.
- `Frame frame_decode (const std::string &data)`
Decodes a string into a `Frame` instance.
- `void frame_process (const std::string &data, Interface interface)`
Executes a command based on the command key and the parameter.
- `Frame frame_build (OperationType operation, uint8_t group, uint8_t command, const std::string &value, const ValueUnit unitType)`
Builds a `Frame` instance based on the execution result, group, command, value, and unit.

6.10.1 Detailed Description

Functions for encoding, decoding and building communication frames.

6.10.2 Function Documentation

6.10.2.1 frame_encode()

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

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

Parameters

<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;

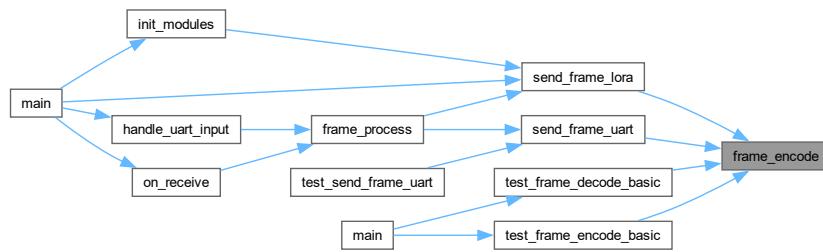
std::string encoded = frame_encode(myFrame);
// encoded will be "KBST;0;GET;1;1;TSBK"
```

Definition at line 37 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.2.2 frame_decode()

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

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

Parameters

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

Returns

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

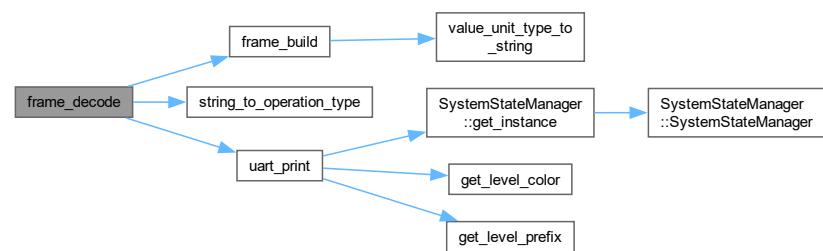
Exceptions

<i>std::runtime_error</i>	if the frame is invalid.
---------------------------	--------------------------

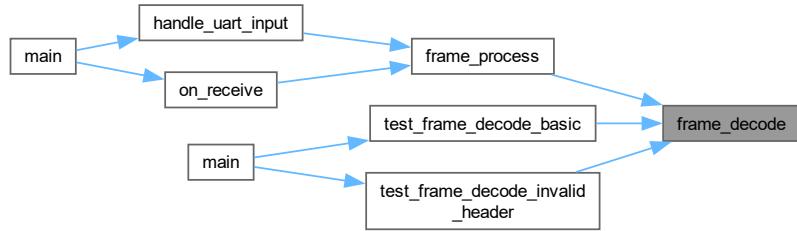
The decoded string is expected to be in the format: FRAME_BEGIN;direction;operationType;group;command;value;unit;FRAME-END

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.2.3 frame_process()

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

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

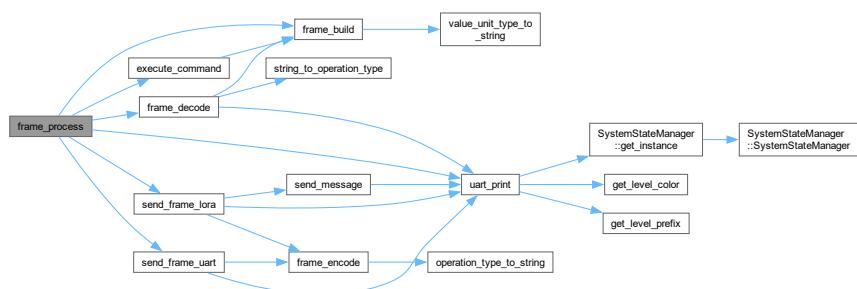
Parameters

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

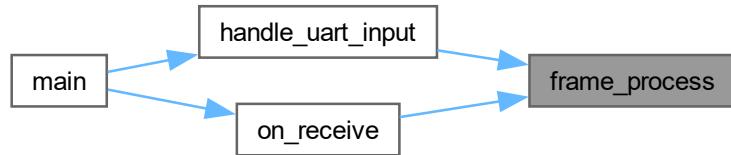
Decodes the frame data, extracts the command key, and executes the corresponding command. Sends the response frame. If an error occurs, an error frame is built and sent.

Definition at line 117 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.10.2.4 frame_build()

```

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

Builds a [Frame](#) instance based on the execution result, group, command, value, and unit.

Parameters

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

Returns

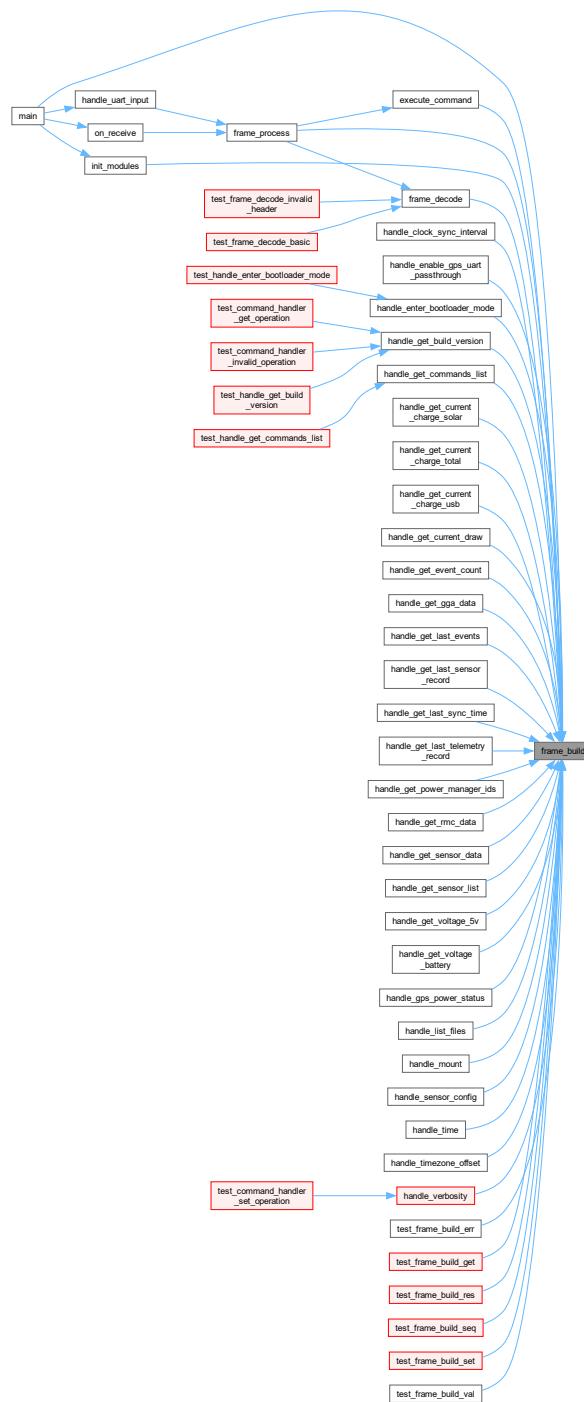
The [Frame](#) instance.

Definition at line 158 of file [frame.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.11 Event Manager

Classes and functions for managing event logging.

Files

- file [event_manager.cpp](#)
Implements the event management system for the Kabisat firmware.

Classes

- class [EventLog](#)
Represents a single event log entry.
- class [EventManager](#)
Manages the event logging system.
- class [EventManagerImpl](#)
Implementation of the [EventManager](#) class.
- class [EventEmitter](#)
Provides a static method for emitting events.

Enumerations

- enum class [EventGroup](#) : uint8_t {
 EventGroup::SYSTEM = 0x00 , EventGroup::POWER = 0x01 , EventGroup::COMMS = 0x02 ,
 EventGroup::GPS = 0x03 ,
 EventGroup::CLOCK = 0x04 }
Represents the group to which an event belongs.
- enum class [SystemEvent](#) : uint8_t {
 SystemEvent::BOOT = 0x01 , SystemEvent::SHUTDOWN = 0x02 , SystemEvent::WATCHDOG_RESET =
 0x03 , SystemEvent::CORE1_START = 0x04 ,
 SystemEvent::CORE1_STOP = 0x05 }
Represents specific system events.
- enum class [PowerEvent](#) : uint8_t {
 PowerEvent::LOW_BATTERY = 0x01 , PowerEvent::OVERCHARGE = 0x02 , PowerEvent::POWER_FALLING
 = 0x03 , PowerEvent::POWER_NORMAL = 0x04 ,
 PowerEvent::SOLAR_ACTIVE = 0x05 , PowerEvent::SOLAR_INACTIVE = 0x06 , PowerEvent::USB_CONNECTED
 = 0x07 , PowerEvent::USB_DISCONNECTED = 0x08 }
Represents specific power-related 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 }
Represents specific communication-related 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 }
Represents specific GPS-related events.
- enum class [ClockEvent](#) : uint8_t { ClockEvent::CHANGED = 0x01 , ClockEvent::GPS_SYNC = 0x02 ,
 ClockEvent::GPS_SYNC_DATA_NOT_READY = 0x03 }
Represents specific clock-related events.

Functions

- class `EventLog __attribute__ ((packed))`
- void `EventManager::log_event (uint8_t group, uint8_t event)`
Logs an event.
- const `EventLog & EventManager::get_event (size_t index) const`
Retrieves an event from the event buffer.

Variables

- volatile `uint16_t eventId = 0`
Global event log ID counter.
- `DS3231 systemClock`
External declaration of the system clock.
- `EventManagerImpl eventManager`
Global instance of the `EventManager` implementation.
- class `EventManager __attribute__`
- `EventManagerImpl eventManager`
Global instance of the `EventManagerImpl` class.

6.11.1 Detailed Description

Classes and functions for managing event logging.

6.11.2 Enumeration Type Documentation

6.11.2.1 EventGroup

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

Represents the group to which an event belongs.

Enumerator

SYSTEM	System events.
POWER	Power-related events.
COMMS	Communication-related events.
GPS	GPS-related events.
CLOCK	Clock-related events.

Definition at line 30 of file `event_manager.h`.

6.11.2.2 SystemEvent

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

Represents specific system 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 47 of file [event_manager.h](#).

6.11.2.3 PowerEvent

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

Represents specific power-related events.

Enumerator

LOW_BATTERY	Low battery event.
OVERCHARGE	Overcharge event.
POWER_FALLING	Power falling event.
POWER_NORMAL	Power normal event.
SOLAR_ACTIVE	Solar charging active event.
SOLAR_INACTIVE	Solar charging inactive event.
USB_CONNECTED	USB connected event.
USB_DISCONNECTED	USB disconnected event.

Definition at line 64 of file [event_manager.h](#).

6.11.2.4 CommsEvent

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

Represents specific communication-related 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 87 of file [event_manager.h](#).

6.11.2.5 GPSEvent

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

Represents specific GPS-related events.

Enumerator

LOCK	GPS lock acquired event.
LOST	GPS lock 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 104 of file [event_manager.h](#).

6.11.2.6 ClockEvent

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

Represents specific clock-related events.

Enumerator

CHANGED	Clock changed event.
GPS_SYNC	Clock synchronized with GPS event.
GPS_SYNC_DATA_NOT_READY	Sync interval but data not ready.

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

6.11.3 Function Documentation

6.11.3.1 __attribute__()

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

6.11.3.2 log_event()

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

Logs an event.

Logs an event to the event buffer.

Parameters

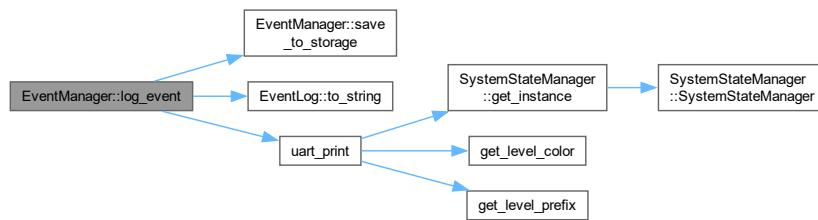
group	The event group.
event	The event identifier.

<i>group</i>	The event group.
<i>event</i>	The event ID.

Logs the event with a timestamp, group, and event ID. Prints the event to the UART, and saves the event to storage if the buffer is full or if it's a power-related event.

Definition at line 45 of file [event_manager.cpp](#).

Here is the call graph for this function:



6.11.3.3 `get_event()`

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

Retrieves an event from the event buffer.

Parameters

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

Returns

A const reference to the `EventLog` at the specified index.

Parameters

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

Returns

A const reference to the `EventLog` at the specified index. Returns an empty event if the index is out of bounds.

Definition at line 83 of file [event_manager.cpp](#).

6.11.4 Variable Documentation

6.11.4.1 eventLogId

```
volatile uint16_t eventLogId = 0
```

Global event log ID counter.

Definition at line 21 of file [event_manager.cpp](#).

6.11.4.2 systemClock

```
DS3231 systemClock [extern]
```

External declaration of the system clock.

6.11.4.3 eventManager [1/2]

```
EventManagerImpl eventManager
```

Global instance of the [EventManager](#) implementation.

Global instance of the [EventManagerImpl](#) class.

Definition at line 33 of file [event_manager.cpp](#).

6.11.4.4 __attribute__

```
class EventManager __attribute__
```

6.11.4.5 eventManager [2/2]

```
EventManagerImpl eventManager [extern]
```

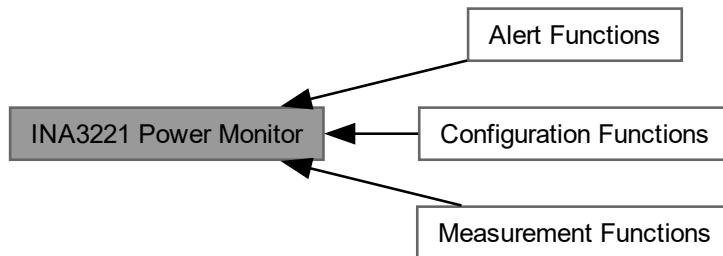
Global instance of the [EventManagerImpl](#) class.

Global instance of the [EventManagerImpl](#) class.

Definition at line 33 of file [event_manager.cpp](#).

6.12 INA3221 Power Monitor

Collaboration diagram for INA3221 Power Monitor:



Topics

- [Configuration Functions](#)
- [Measurement Functions](#)
- [Alert Functions](#)

6.12.1 Detailed Description

6.12.2 Configuration Functions

Collaboration diagram for Configuration Functions:



Functions

- [`INA3221::INA3221 \(ina3221_addr_t addr, i2c_inst_t *i2c\)`](#)
Constructor for `INA3221` class.
- [`bool INA3221::begin \(\)`](#)
Initialize the `INA3221` device.
- [`void INA3221::reset \(\)`](#)
Reset the `INA3221` to default settings.
- [`uint16_t INA3221::get_manufacturer_id \(\)`](#)

- `uint16_t INA3221::get_die_id ()`

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

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

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

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

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

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

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

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

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

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

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

Set shunt voltage conversion time.

6.12.2.1 Detailed Description

Functions for configuring the [INA3221](#) device

6.12.2.2 Function Documentation

6.12.2.2.1 [INA3221\(\)](#)

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

Constructor for [INA3221](#) class.

Parameters

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

Definition at line [46](#) of file [INA3221.cpp](#).

6.12.2.2.2 begin()

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

Initialize the [INA3221](#) device.

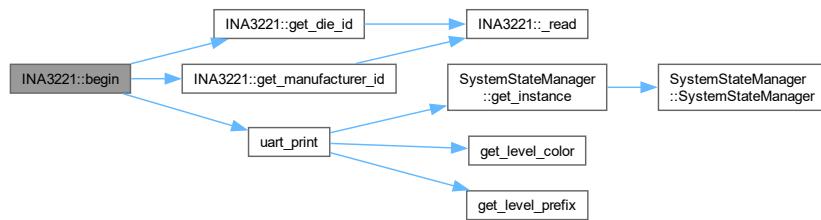
Returns

true if initialization successful, false otherwise

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

Definition at line 56 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.3 reset()

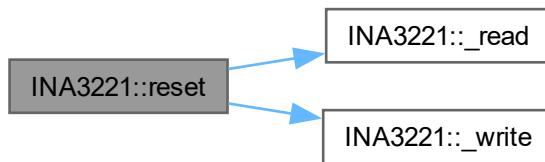
```
void INA3221::reset ()
```

Reset the [INA3221](#) to default settings.

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

Definition at line 90 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.4 get_manufacturer_id()

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

Get the manufacturer ID of the device.

Returns

16-bit manufacturer ID (should be 0x5449)

Definition at line 104 of file [INA3221.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.12.2.2.5 get_die_id()

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

Get the die ID of the device.

Returns

16-bit die ID (should be 0x3220)

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.12.2.2.6 `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 129 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.7 set_mode_power_down()

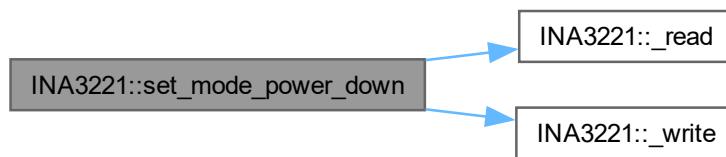
```
void INA3221::set_mode_power_down ()
```

Set device to power-down mode.

Disables bus voltage and continuous measurements

Definition at line 143 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.8 set_mode_continuous()

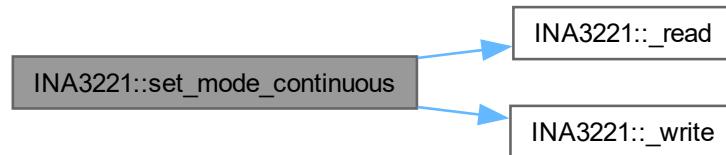
```
void INA3221::set_mode_continuous ()
```

Set device to continuous measurement mode.

Enables continuous measurement of bus voltage and shunt voltage

Definition at line 158 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.9 set_mode_triggered()

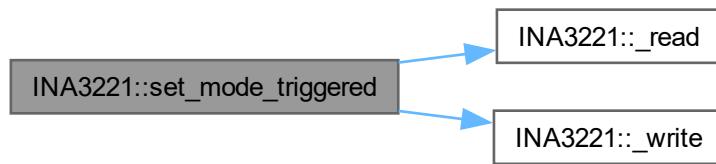
```
void INA3221::set_mode_triggered ()
```

Set device to triggered measurement mode.

Disables continuous measurements, requiring manual triggers

Definition at line 172 of file [INA3221.cpp](#).

Here is the call graph for this function:



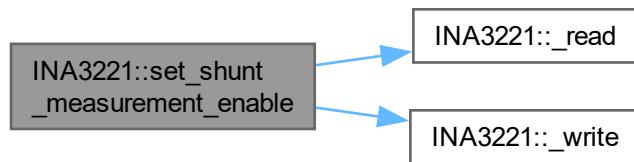
6.12.2.2.10 set_shunt_measurement_enable()

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

Enable shunt voltage measurements.

Definition at line 185 of file [INA3221.cpp](#).

Here is the call graph for this function:



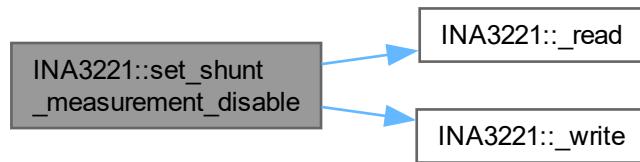
6.12.2.2.11 set_shunt_measurement_disable()

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

Disable shunt voltage measurements.

Definition at line 198 of file [INA3221.cpp](#).

Here is the call graph for this function:



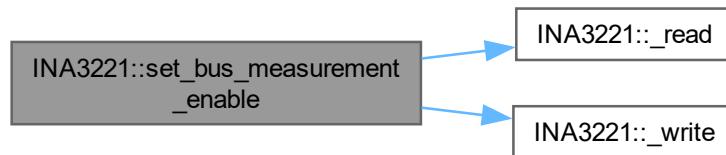
6.12.2.2.12 set_bus_measurement_enable()

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

Enable bus voltage measurements.

Definition at line 211 of file [INA3221.cpp](#).

Here is the call graph for this function:



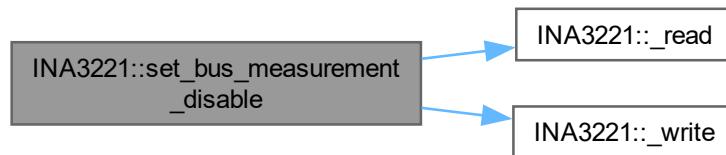
6.12.2.2.13 set_bus_measurement_disable()

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

Disable bus voltage measurements.

Definition at line 224 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.14 set_averaging_mode()

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

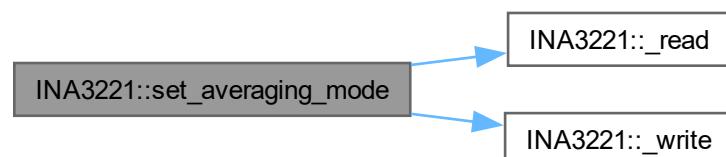
Set the averaging mode for measurements.

Parameters

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

Definition at line 238 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.2.2.15 set_bus_conversion_time()

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

Set bus voltage conversion time.

Parameters

<i>convTime</i>	Conversion time setting
-----------------	-------------------------

Definition at line 252 of file [INA3221.cpp](#).

Here is the call graph for this function:

**6.12.2.2.16 set_shunt_conversion_time()**

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

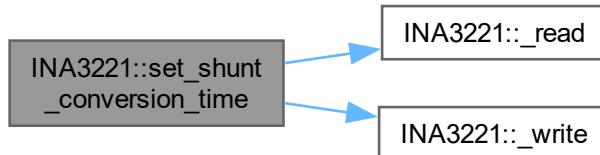
Set shunt voltage conversion time.

Parameters

<i>convTime</i>	Conversion time setting
-----------------	-------------------------

Definition at line 266 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.3 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.

6.12.3.1 Detailed Description

Functions for reading voltage, current and power measurements

6.12.3.2 Function Documentation

6.12.3.2.1 `get_shunt_voltage()`

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

Get shunt voltage for a specific channel.

Parameters

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

Returns

Shunt voltage in microvolts (μ V)

Definition at line 282 of file [INA3221.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.12.3.2.2 get_current_ma()

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

Get current for a specific channel.

Parameters

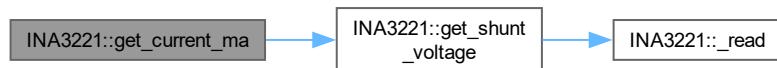
<i>channel</i>	Channel number (1-3)
----------------	----------------------

Returns

Current in millamps (mA)

Definition at line 314 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.3.2.3 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 330 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.4 Alert Functions

Collaboration diagram for Alert Functions:



Functions

- void [INA3221::set_warn_alert_limit](#) (ina3221_ch_t channel, float voltage_v)
Set warning alert voltage threshold for a channel.
- void [INA3221::set_crit_alert_limit](#) (ina3221_ch_t channel, float voltage_v)
Set critical alert voltage threshold for a channel.
- void [INA3221::set_power_valid_limit](#) (float voltage_upper_v, float voltage_lower_v)
Set power valid voltage range.

- void `INA3221::enable_alerts ()`
Enable all alert functions.
- bool `INA3221::get_warn_alert (ina3221_ch_t channel)`
Get warning alert status for a channel.
- bool `INA3221::get_crit_alert (ina3221_ch_t channel)`
Get critical alert status for a channel.
- bool `INA3221::get_power_valid_alert ()`
Get power valid alert status.
- void `INA3221::set_alert_latch (bool enable)`
Set alert latch mode.

6.12.4.1 Detailed Description

Functions for configuring and reading alert conditions

6.12.4.2 Function Documentation

6.12.4.2.1 `set_warn_alert_limit()`

```
void INA3221::set_warn_alert_limit (
    ina3221_ch_t channel,
    float voltage_v)
```

Set warning alert voltage threshold for a channel.

Parameters

<code>channel</code>	Channel number (1-3)
<code>voltage_v</code>	Voltage threshold in volts

Definition at line 360 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.4.2.2 `set_crit_alert_limit()`

```
void INA3221::set_crit_alert_limit (
    ina3221_ch_t channel,
    float voltage_v)
```

Set critical alert voltage threshold for a channel.

Parameters

<i>channel</i>	Channel number (1-3)
<i>voltage_v</i>	Voltage threshold in volts

Definition at line 385 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.4.2.3 set_power_valid_limit()

```
void INA3221::set_power_valid_limit (
    float voltage_upper_v,
    float voltage_lower_v)
```

Set power valid voltage range.

Parameters

<i>voltage_upper_v</i>	Upper voltage threshold in volts
<i>voltage_lower_v</i>	Lower voltage threshold in volts

Definition at line 410 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.4.2.4 enable_alerts()

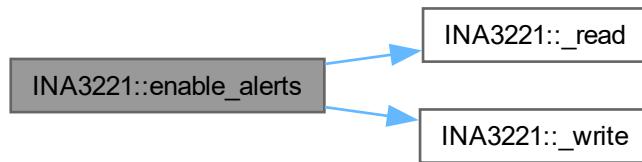
```
void INA3221::enable_alerts ()
```

Enable all alert functions.

Enables warning alerts, critical alerts, and power valid alerts for all channels

Definition at line 426 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.4.2.5 get_warn_alert()

```
bool INA3221::get_warn_alert (
    ina3221_ch_t channel)
```

Get warning alert status for a channel.

Parameters

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

Returns

true if warning alert is active, false otherwise

Definition at line 448 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.12.4.2.6 `get_crit_alert()`

```
bool INA3221::get_crit_alert (
    ina3221_ch_t channel)
```

Get critical alert status for a channel.

Parameters

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

Returns

true if critical alert is active, false otherwise

Definition at line 467 of file [INA3221.cpp](#).

Here is the call graph for this function:

**6.12.4.2.7 get_power_valid_alert()**

```
bool INA3221::get_power_valid_alert ()
```

Get power valid alert status.

Returns

true if power valid alert is active, false otherwise

Definition at line 485 of file [INA3221.cpp](#).

Here is the call graph for this function:

**6.12.4.2.8 set_alert_latch()**

```
void INA3221::set_alert_latch (
    bool enable)
```

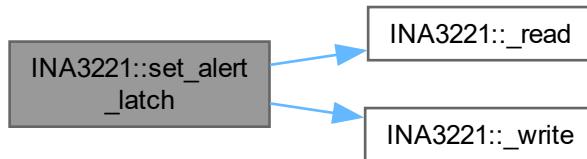
Set alert latch mode.

Parameters

<code>enable</code>	true to enable alert latching, false for transparent alerts
---------------------	---

Definition at line 497 of file [INA3221.cpp](#).

Here is the call graph for this function:



6.13 Telemetry Manager

Classes

- struct [TelemetryRecord](#)
Structure representing a single telemetry data point.
- struct [SensorDataRecord](#)

Functions

- bool [telemetry_init \(\)](#)
Initialize the telemetry system.
- bool [collect_telemetry \(\)](#)
Collect telemetry data from sensors and power subsystems.
- bool [flush_telemetry \(\)](#)
Save buffered telemetry data to storage.
- bool [flush_sensor_data \(\)](#)
Save buffered sensor data to storage.
- bool [is_telemetry_collection_time \(uint32_t current_time, uint32_t &last_collection_time\)](#)
Check if it's time to collect telemetry based on interval.
- bool [is_telemetry_flush_time \(uint32_t &collection_counter\)](#)
Check if it's time to flush telemetry buffer based on count.
- uint32_t [get_telemetry_sample_interval \(\)](#)
Get the current sample interval in milliseconds.
- void [set_telemetry_sample_interval \(uint32_t interval_ms\)](#)
Set the telemetry sample interval.
- uint32_t [get_telemetry_flush_threshold \(\)](#)
Get the number of records before flushing to storage.
- void [set_telemetry_flush_threshold \(uint32_t records\)](#)
Set the number of records before flushing to storage.
- std::string [get_last_telemetry_record_csv \(\)](#)
Gets the last telemetry record as a CSV string.
- std::string [get_last_sensor_record_csv \(\)](#)
Gets the last sensor data record as a CSV string.

Variables

- `constexpr int TELEMETRY_BUFFER_SIZE = 20`
Circular buffer for telemetry records.
- `TelemetryRecord telemetry_buffer [TELEMETRY_BUFFER_SIZE]`
Circular buffer for telemetry records.
- `size_t telemetry_buffer_count`
- `size_t telemetry_buffer_write_index`
- `SensorDataRecord sensor_data_buffer [TELEMETRY_BUFFER_SIZE]`
Circular buffer for sensor data records.
- `size_t sensor_data_buffer_count`
- `size_t sensor_data_buffer_write_index`
- `mutex_t telemetry_mutex`
Mutex for thread-safe access to the telemetry buffer.
- `const int TELEMETRY_BUFFER_SIZE`
Circular buffer for telemetry records.

6.13.1 Detailed Description

6.13.2 Function Documentation

6.13.2.1 `telemetry_init()`

```
bool telemetry_init ()
```

Initialize the telemetry system.

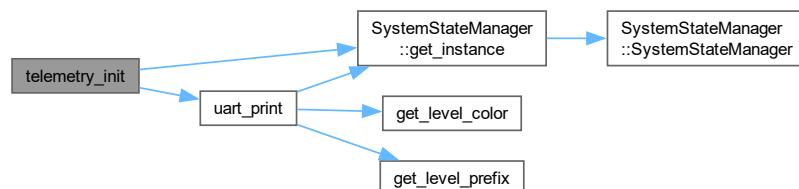
Returns

True if initialization was successful

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

Definition at line 77 of file `telemetry_manager.cpp`.

Here is the call graph for this function:



Here is the caller graph for this function:



6.13.2.2 collect_telemetry()

```
bool collect_telemetry ()
```

Collect telemetry data from sensors and power subsystems.

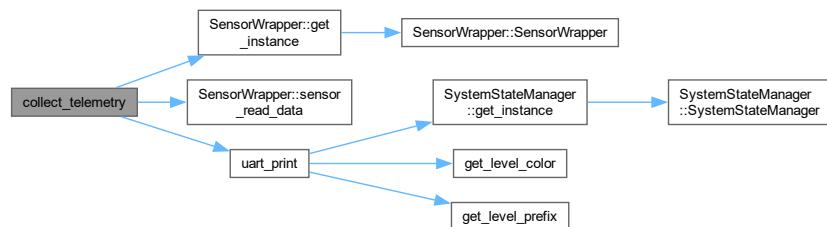
Returns

True if data was successfully collected

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.13.2.3 flush_telemetry()

```
bool flush_telemetry ()
```

Save buffered telemetry data to storage.

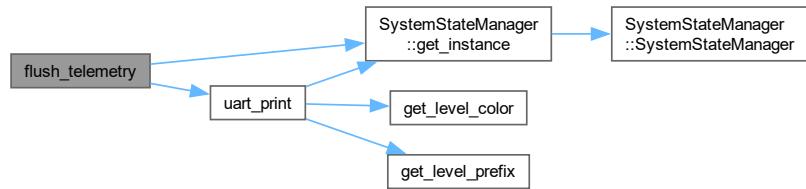
Returns

True if data was successfully saved

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

Definition at line 195 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.13.2.4 flush_sensor_data()

```
bool flush_sensor_data ()
```

Save buffered sensor data to storage.

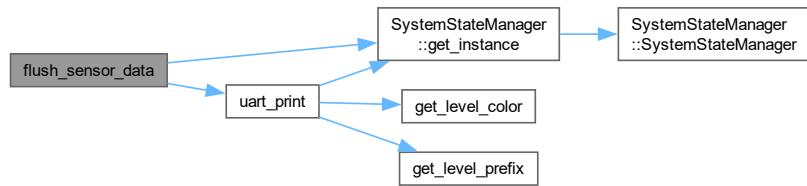
Returns

True if data was successfully saved

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

Definition at line 238 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**6.13.2.5 is_telemetry_collection_time()**

```

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

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

Parameters

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

Returns

True if collection interval has passed

Updates last_collection_time if the interval has passed

Definition at line 279 of file [telemetry_manager.cpp](#).

Here is the caller graph for this function:



6.13.2.6 is_telemetry_flush_time()

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

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

Parameters

<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 287 of file [telemetry_manager.cpp](#).

Here is the caller graph for this function:



6.13.2.7 get_telemetry_sample_interval()

```
uint32_t get_telemetry_sample_interval ()
```

Get the current sample interval in milliseconds.

Returns

Sample interval in milliseconds

Definition at line 295 of file [telemetry_manager.cpp](#).

6.13.2.8 set_telemetry_sample_interval()

```
void set_telemetry_sample_interval (
    uint32_t interval_ms)
```

Set the telemetry sample interval.

Parameters

<i>interval_ms</i>	New interval in milliseconds
--------------------	------------------------------

Sets a minimum bound of 100ms to prevent excessive sampling

Definition at line 299 of file [telemetry_manager.cpp](#).

6.13.2.9 get_telemetry_flush_threshold()

```
uint32_t get_telemetry_flush_threshold ()
```

Get the number of records before flushing to storage.

Returns

Number of records in flush threshold

Definition at line 305 of file [telemetry_manager.cpp](#).

6.13.2.10 set_telemetry_flush_threshold()

```
void set_telemetry_flush_threshold (
    uint32_t records)
```

Set the number of records before flushing to storage.

Parameters

<i>records</i>	Number of records in flush threshold
----------------	--------------------------------------

Sets reasonable bounds (1-100) for the threshold value

Definition at line 309 of file [telemetry_manager.cpp](#).

6.13.2.11 `get_last_telemetry_record_csv()`

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

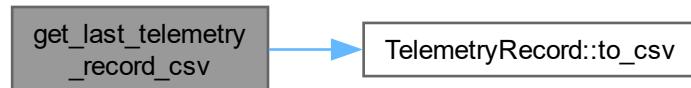
Gets the last telemetry record as a CSV string.

Returns

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

Definition at line 315 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.13.2.12 `get_last_sensor_record_csv()`

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

Gets the last sensor data record as a CSV string.

Returns

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

Definition at line 332 of file [telemetry_manager.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.13.3 Variable Documentation

6.13.3.1 TELEMETRY_BUFFER_SIZE [1/2]

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

Circular buffer for telemetry records.

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

6.13.3.2 telemetry_buffer

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

Circular buffer for telemetry records.

Definition at line 61 of file [telemetry_manager.cpp](#).

6.13.3.3 telemetry_buffer_count

```
size_t telemetry_buffer_count [extern]
```

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

6.13.3.4 `telemetry_buffer_write_index`

```
size_t telemetry_buffer_write_index [extern]
```

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

6.13.3.5 `sensor_data_buffer`

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

Circular buffer for sensor data records.

Definition at line 68 of file [telemetry_manager.cpp](#).

6.13.3.6 `sensor_data_buffer_count`

```
size_t sensor_data_buffer_count [extern]
```

Definition at line 69 of file [telemetry_manager.cpp](#).

6.13.3.7 `sensor_data_buffer_write_index`

```
size_t sensor_data_buffer_write_index [extern]
```

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

6.13.3.8 `telemetry_mutex`

```
mutex_t telemetry_mutex [extern]
```

Mutex for thread-safe access to the telemetry buffer.

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

6.13.3.9 `TELEMETRY_BUFFER_SIZE` [2/2]

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

Circular buffer for telemetry records.

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

Chapter 7

Class Documentation

7.1 BH1750 Class Reference

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

Public Types

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

Public Member Functions

- `BH1750` (`uint8_t` `addr`=0x23)
- bool `begin` (`Mode mode`=`Mode::CONTINUOUS_HIGH_RES_MODE`)
- void `configure` (`Mode mode`)
- float `get_light_level` ()

Private Member Functions

- void `write8` (`uint8_t data`)

Private Attributes

- `uint8_t _i2c_addr`

7.1.1 Detailed Description

Definition at line 12 of file `BH1750.h`.

7.1.2 Member Enumeration Documentation

7.1.2.1 Mode

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

Enumerator

UNCONFIGURED_POWER_DOWN	
POWER_ON	
RESET	
CONTINUOUS_HIGH_RES_MODE	
CONTINUOUS_HIGH_RES_MODE_2	
CONTINUOUS_LOW_RES_MODE	
ONE_TIME_HIGH_RES_MODE	
ONE_TIME_HIGH_RES_MODE_2	
ONE_TIME_LOW_RES_MODE	

Definition at line 15 of file [BH1750.h](#).

7.1.3 Constructor & Destructor Documentation

7.1.3.1 BH1750()

```
BH1750::BH1750 (
    uint8_t addr = 0x23)
```

Definition at line 6 of file [BH1750.cpp](#).

7.1.4 Member Function Documentation

7.1.4.1 begin()

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

Definition at line 8 of file [BH1750.cpp](#).

Here is the call graph for this function:



7.1.4.2 configure()

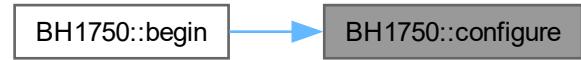
```
void BH1750::configure (
    Mode mode)
```

Definition at line 21 of file [BH1750.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.4.3 get_light_level()

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

Definition at line 39 of file [BH1750.cpp](#).

7.1.4.4 write8()

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

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

Here is the caller graph for this function:



7.1.5 Member Data Documentation

7.1.5.1 _i2c_addr

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

Definition at line 34 of file [BH1750.h](#).

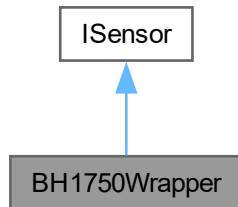
The documentation for this class was generated from the following files:

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

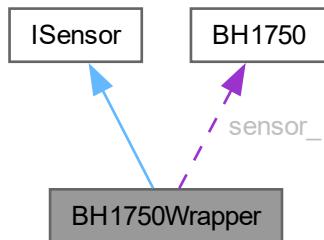
7.2 BH1750Wrapper Class Reference

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

Inheritance diagram for BH1750Wrapper:



Collaboration diagram for BH1750Wrapper:



Public Member Functions

- `BH1750Wrapper ()`
- `int get_i2c_addr ()`
- `bool init () override`
- `float read_data (SensorDataTypelIdentifier type) override`
- `bool is_initialized () const override`
- `SensorType get_type () const override`
- `bool configure (const std::map< std::string, std::string > &config)`
- `uint8_t get_address () const override`

Public Member Functions inherited from `ISensor`

- `virtual ~ISensor ()=default`

Private Attributes

- `BH1750 sensor_`
- `bool initialized_ = false`

7.2.1 Detailed Description

Definition at line 9 of file `BH1750_WRAPPER.h`.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 `BH1750Wrapper()`

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

Definition at line 6 of file `BH1750_WRAPPER.cpp`.

7.2.3 Member Function Documentation

7.2.3.1 `get_i2c_addr()`

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

7.2.3.2 `init()`

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

Implements `ISensor`.

Definition at line 10 of file `BH1750_WRAPPER.cpp`.

7.2.3.3 `read_data()`

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

Implements [ISensor](#).

Definition at line 15 of file [BH1750_WRAPPER.cpp](#).

7.2.3.4 `is_initialized()`

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

Implements [ISensor](#).

Definition at line 22 of file [BH1750_WRAPPER.cpp](#).

7.2.3.5 `get_type()`

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

Implements [ISensor](#).

Definition at line 26 of file [BH1750_WRAPPER.cpp](#).

7.2.3.6 `configure()`

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

Implements [ISensor](#).

Definition at line 30 of file [BH1750_WRAPPER.cpp](#).

7.2.3.7 `get_address()`

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

Implements [ISensor](#).

Definition at line 24 of file [BH1750_WRAPPER.h](#).

7.2.4 Member Data Documentation

7.2.4.1 `sensor_`

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

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

7.2.4.2 initialized_

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

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

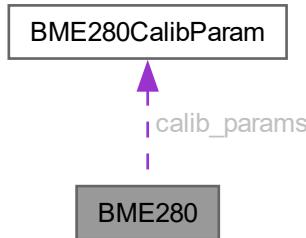
The documentation for this class was generated from the following files:

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

7.3 BME280 Class Reference

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

Collaboration diagram for BME280:



Public Member Functions

- [BME280 \(i2c_inst_t *i2cPort, uint8_t address=ADDR_SDO_LOW\)](#)
- bool [init \(\)](#)
- void [reset \(\)](#)
- bool [read_raw_all \(int32_t *temperature, int32_t *pressure, int32_t *humidity\)](#)
- float [convert_temperature \(int32_t temp_raw\) const](#)
- float [convert_pressure \(int32_t pressure_raw\) const](#)
- float [convert_humidity \(int32_t humidity_raw\) const](#)

Static Public Attributes

- static constexpr uint8_t [ADDR_SDO_LOW](#) = 0x76
- static constexpr uint8_t [ADDR_SDO_HIGH](#) = 0x77

Private Member Functions

- bool [configure_sensor \(\)](#)
- bool [get_calibration_parameters \(\)](#)

Private Attributes

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

Static Private Attributes

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

7.3.1 Detailed Description

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

7.3.2 Constructor & Destructor Documentation

7.3.2.1 BME280()

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

Definition at line 14 of file [BME280.cpp](#).

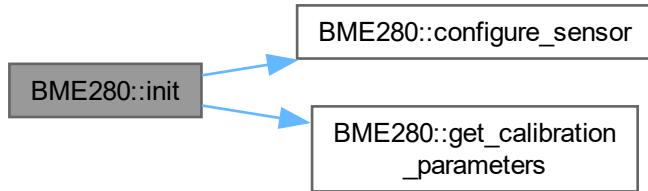
7.3.3 Member Function Documentation

7.3.3.1 init()

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

Definition at line 18 of file [BME280.cpp](#).

Here is the call graph for this function:



7.3.3.2 reset()

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

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

7.3.3.3 read_raw_all()

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

Definition at line 68 of file [BME280.cpp](#).

7.3.3.4 convert_temperature()

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

Definition at line 101 of file [BME280.cpp](#).

7.3.3.5 convert_pressure()

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

Definition at line 110 of file [BME280.cpp](#).

7.3.3.6 convert_humidity()

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

Definition at line 131 of file [BME280.cpp](#).

7.3.3.7 configure_sensor()

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

Definition at line 201 of file [BME280.cpp](#).

Here is the caller graph for this function:



7.3.3.8 get_calibration_parameters()

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

Definition at line 143 of file [BME280.cpp](#).

Here is the caller graph for this function:



7.3.4 Member Data Documentation

7.3.4.1 ADDR_SDO_LOW

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

Definition at line 41 of file [BME280.h](#).

7.3.4.2 ADDR_SDO_HIGH

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

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

7.3.4.3 i2c_port

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

Definition at line 69 of file [BME280.h](#).

7.3.4.4 device_addr

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

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

7.3.4.5 calib_params

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

Definition at line 73 of file [BME280.h](#).

7.3.4.6 initialized_

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

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

7.3.4.7 t_fine

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

Definition at line 79 of file [BME280.h](#).

7.3.4.8 REG_CONFIG

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

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

7.3.4.9 REG_CTRL_MEAS

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

Definition at line [83](#) of file [BME280.h](#).

7.3.4.10 REG_CTRL_HUM

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

Definition at line [84](#) of file [BME280.h](#).

7.3.4.11 REG_RESET

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

Definition at line [85](#) of file [BME280.h](#).

7.3.4.12 REG_PRESSURE_MSB

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

Definition at line [87](#) of file [BME280.h](#).

7.3.4.13 REG_TEMPERATURE_MSB

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

Definition at line [88](#) of file [BME280.h](#).

7.3.4.14 REG_HUMIDITY_MSB

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

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

7.3.4.15 REG_DIG_T1_LSB

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

Definition at line [92](#) of file [BME280.h](#).

7.3.4.16 REG_DIG_T1_MSB

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

Definition at line 93 of file [BME280.h](#).

7.3.4.17 REG_DIG_T2_LSB

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

Definition at line 94 of file [BME280.h](#).

7.3.4.18 REG_DIG_T2_MSB

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

Definition at line 95 of file [BME280.h](#).

7.3.4.19 REG_DIG_T3_LSB

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

Definition at line 96 of file [BME280.h](#).

7.3.4.20 REG_DIG_T3_MSB

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

Definition at line 97 of file [BME280.h](#).

7.3.4.21 REG_DIG_P1_LSB

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

Definition at line 99 of file [BME280.h](#).

7.3.4.22 REG_DIG_P1_MSB

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

Definition at line 100 of file [BME280.h](#).

7.3.4.23 REG_DIG_P2_LSB

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

Definition at line 101 of file [BME280.h](#).

7.3.4.24 REG_DIG_P2_MSB

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

Definition at line 102 of file [BME280.h](#).

7.3.4.25 REG_DIG_P3_LSB

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

Definition at line 103 of file [BME280.h](#).

7.3.4.26 REG_DIG_P3_MSB

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

Definition at line 104 of file [BME280.h](#).

7.3.4.27 REG_DIG_P4_LSB

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

Definition at line 105 of file [BME280.h](#).

7.3.4.28 REG_DIG_P4_MSB

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

Definition at line 106 of file [BME280.h](#).

7.3.4.29 REG_DIG_P5_LSB

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

Definition at line 107 of file [BME280.h](#).

7.3.4.30 REG_DIG_P5_MSB

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

Definition at line 108 of file [BME280.h](#).

7.3.4.31 REG_DIG_P6_LSB

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

Definition at line 109 of file [BME280.h](#).

7.3.4.32 REG_DIG_P6_MSB

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

Definition at line 110 of file [BME280.h](#).

7.3.4.33 REG_DIG_P7_LSB

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

Definition at line 111 of file [BME280.h](#).

7.3.4.34 REG_DIG_P7_MSB

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

Definition at line 112 of file [BME280.h](#).

7.3.4.35 REG_DIG_P8_LSB

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

Definition at line 113 of file [BME280.h](#).

7.3.4.36 REG_DIG_P8_MSB

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

Definition at line 114 of file [BME280.h](#).

7.3.4.37 REG_DIG_P9_LSB

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

Definition at line 115 of file [BME280.h](#).

7.3.4.38 REG_DIG_P9_MSB

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

Definition at line 116 of file [BME280.h](#).

7.3.4.39 REG_DIG_H1

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

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

7.3.4.40 REG_DIG_H2

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

Definition at line 120 of file [BME280.h](#).

7.3.4.41 REG_DIG_H3

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

Definition at line 121 of file [BME280.h](#).

7.3.4.42 REG_DIG_H4

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

Definition at line 122 of file [BME280.h](#).

7.3.4.43 REG_DIG_H5

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

Definition at line 123 of file [BME280.h](#).

7.3.4.44 REG_DIG_H6

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

Definition at line 124 of file [BME280.h](#).

7.3.4.45 NUM_CALIB_PARAMS

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

Definition at line 127 of file [BME280.h](#).

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

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

7.4 BME280CalibParam Struct Reference

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

Public Attributes

- `uint16_t dig_t1`
- `int16_t dig_t2`
- `int16_t dig_t3`
- `uint16_t dig_p1`
- `int16_t dig_p2`
- `int16_t dig_p3`
- `int16_t dig_p4`
- `int16_t dig_p5`
- `int16_t dig_p6`
- `int16_t dig_p7`
- `int16_t dig_p8`
- `int16_t dig_p9`
- `uint8_t dig_h1`
- `int16_t dig_h2`
- `uint8_t dig_h3`
- `int16_t dig_h4`
- `int16_t dig_h5`
- `int8_t dig_h6`

7.4.1 Detailed Description

Definition at line 11 of file [BME280.h](#).

7.4.2 Member Data Documentation

7.4.2.1 `dig_t1`

```
uint16_t BME280CalibParam::dig_t1
```

Definition at line 13 of file [BME280.h](#).

7.4.2.2 `dig_t2`

```
int16_t BME280CalibParam::dig_t2
```

Definition at line 14 of file [BME280.h](#).

7.4.2.3 `dig_t3`

```
int16_t BME280CalibParam::dig_t3
```

Definition at line 15 of file [BME280.h](#).

7.4.2.4 `dig_p1`

```
uint16_t BME280CalibParam::dig_p1
```

Definition at line 18 of file [BME280.h](#).

7.4.2.5 `dig_p2`

```
int16_t BME280CalibParam::dig_p2
```

Definition at line 19 of file [BME280.h](#).

7.4.2.6 `dig_p3`

```
int16_t BME280CalibParam::dig_p3
```

Definition at line 20 of file [BME280.h](#).

7.4.2.7 `dig_p4`

```
int16_t BME280CalibParam::dig_p4
```

Definition at line 21 of file [BME280.h](#).

7.4.2.8 `dig_p5`

```
int16_t BME280CalibParam::dig_p5
```

Definition at line 22 of file [BME280.h](#).

7.4.2.9 `dig_p6`

```
int16_t BME280CalibParam::dig_p6
```

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

7.4.2.10 `dig_p7`

```
int16_t BME280CalibParam::dig_p7
```

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

7.4.2.11 `dig_p8`

```
int16_t BME280CalibParam::dig_p8
```

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

7.4.2.12 dig_p9

```
int16_t BME280CalibParam::dig_p9
```

Definition at line 26 of file [BME280.h](#).

7.4.2.13 dig_h1

```
uint8_t BME280CalibParam::dig_h1
```

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

7.4.2.14 dig_h2

```
int16_t BME280CalibParam::dig_h2
```

Definition at line 30 of file [BME280.h](#).

7.4.2.15 dig_h3

```
uint8_t BME280CalibParam::dig_h3
```

Definition at line 31 of file [BME280.h](#).

7.4.2.16 dig_h4

```
int16_t BME280CalibParam::dig_h4
```

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

7.4.2.17 dig_h5

```
int16_t BME280CalibParam::dig_h5
```

Definition at line 33 of file [BME280.h](#).

7.4.2.18 dig_h6

```
int8_t BME280CalibParam::dig_h6
```

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

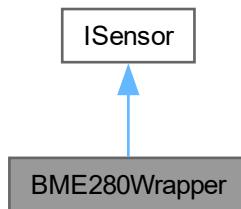
The documentation for this struct was generated from the following file:

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

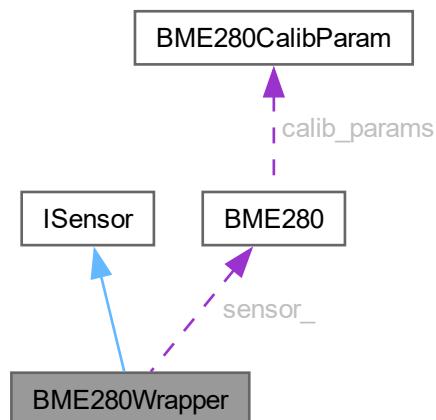
7.5 BME280Wrapper Class Reference

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

Inheritance diagram for BME280Wrapper:



Collaboration diagram for BME280Wrapper:



Public Member Functions

- `BME280Wrapper (i2c_inst_t *i2c)`
- `bool init () override`
- `float read_data (SensorDataTypelIdentifier type) override`
- `bool is_initialized () const override`
- `SensorType get_type () const override`
- `bool configure (const std::map< std::string, std::string > &config) override`
- `uint8_t get_address () const override`

Public Member Functions inherited from [ISensor](#)

- virtual [~ISensor](#) ()=default

Private Attributes

- [BME280 sensor_](#)
- bool [initialized_](#) = false

7.5.1 Detailed Description

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

7.5.2 Constructor & Destructor Documentation

7.5.2.1 [BME280Wrapper\(\)](#)

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

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

7.5.3 Member Function Documentation

7.5.3.1 [init\(\)](#)

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

Implements [ISensor](#).

Definition at line 5 of file [BME280_WRAPPER.cpp](#).

7.5.3.2 [read_data\(\)](#)

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

Implements [ISensor](#).

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

7.5.3.3 [is_initialized\(\)](#)

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

Implements [ISensor](#).

Definition at line 26 of file [BME280_WRAPPER.cpp](#).

7.5.3.4 `get_type()`

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

Implements [ISensor](#).

Definition at line 30 of file [BME280_WRAPPER.cpp](#).

7.5.3.5 `configure()`

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

Implements [ISensor](#).

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

7.5.3.6 `get_address()`

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

Implements [ISensor](#).

Definition at line 22 of file [BME280_WRAPPER.h](#).

7.5.4 Member Data Documentation

7.5.4.1 `sensor_`

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

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

7.5.4.2 `initialized_`

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

Definition at line 11 of file [BME280_WRAPPER.h](#).

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

- lib/sensors/BME280/[BME280_WRAPPER.h](#)
- lib/sensors/BME280/[BME280_WRAPPER.cpp](#)

7.6 INA3221::conf_reg_t Struct Reference

Configuration register bit fields.

Public Attributes

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

7.6.1 Detailed Description

Configuration register bit fields.

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

7.6.2 Member Data Documentation

7.6.2.1 mode_shunt_en

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

Definition at line 102 of file [INA3221.h](#).

7.6.2.2 mode_bus_en

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

Definition at line 103 of file [INA3221.h](#).

7.6.2.3 mode_continious_en

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

Definition at line 104 of file [INA3221.h](#).

7.6.2.4 shunt_conv_time

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

Definition at line 105 of file [INA3221.h](#).

7.6.2.5 bus_conv_time

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

Definition at line 106 of file [INA3221.h](#).

7.6.2.6 avg_mode

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

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

7.6.2.7 ch3_en

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

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

7.6.2.8 ch2_en

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

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

7.6.2.9 ch1_en

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

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

7.6.2.10 reset

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

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

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

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

7.7 DS3231 Class Reference

Class for interfacing with the [DS3231](#) real-time clock.

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

Public Member Functions

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

Private Member Functions

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

Private Attributes

- `i2c_inst_t * i2c`
`Pointer to I2C instance.`
- `uint8_t ds3231_addr`
`I2C address of DS3231.`
- `recursive_mutex_t clock_mutex_`
`Mutex for thread-safe I2C access.`
- `int16_t timezone_offset_minutes_ = 60`
`Timezone offset in minutes, default: UTC.`
- `uint32_t sync_interval_minutes_ = 1440`
`Sync interval in minutes, default: 24 hours.`
- `time_t last_sync_time_ = 0`
`Last sync timestamp, 0 = never synced.`

7.7.1 Detailed Description

Class for interfacing with the [DS3231](#) real-time clock.

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

Definition at line 108 of file [DS3231.h](#).

7.7.2 Constructor & Destructor Documentation

7.7.2.1 DS3231()

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

Constructor for the [DS3231](#) class.

Parameters

in	<code>i2c_instance</code>	Pointer to the I2C instance to use
in	<code>i2c_instance</code>	Pointer to the <code>i2c_inst_t</code> structure representing the I ² C port

Initializes a [DS3231](#) RTC controller object with the specified I²C port and default device address. This constructor also initializes the clock mutex for thread-safe I²C access.

Definition at line 16 of file [DS3231.cpp](#).

7.7.3 Member Function Documentation

7.7.3.1 set_time()

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

Sets the time on the [DS3231](#) clock.

Sets the time on the [DS3231](#) RTC module.

Parameters

in	<i>data</i>	Pointer to a ds3231_data_t structure with time information
----	-------------	--

Returns

0 on success, -1 on failure

Parameters

in	<i>data</i>	Pointer to a ds3231_data_t structure containing the time to set
----	-------------	---

Returns

0 on success, -1 on failure

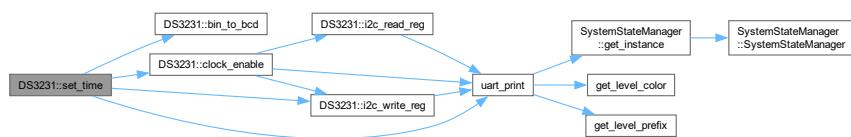
Writes time and date data to the [DS3231](#) module. The function performs input validation to ensure that the provided values are within valid ranges. The time values are converted from binary to BCD format before being written to the device registers.

Note

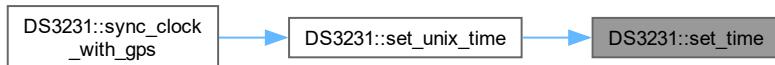
The [ds3231_data_t](#) structure must contain valid values for seconds, minutes, hours, day, date, month, year, and century.

Definition at line 33 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.7.3.2 `get_time()`**

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

Gets the current time from the [DS3231](#) clock.

Reads the current time from the [DS3231](#) RTC module.

Parameters

<code>out</code>	<code>data</code>	Pointer to a ds3231_data_t structure to store time information
------------------	-------------------	--

Returns

0 on success, -1 on failure

Parameters

<code>out</code>	<code>data</code>	Pointer to a ds3231_data_t structure to store the read time
------------------	-------------------	---

Returns

0 on success, -1 on failure

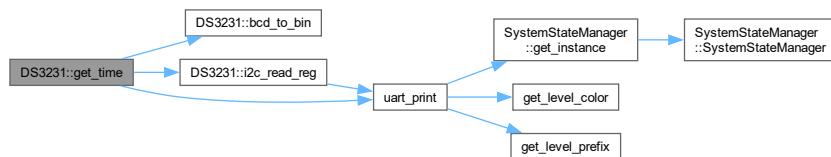
Reads the time and date registers from the [DS3231](#) and stores the decoded values in the provided data structure. The BCD values from the registers are converted to binary format. The function performs validation on the read values to ensure they are within valid ranges.

Note

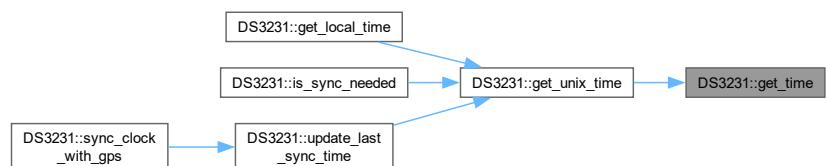
The function logs debug information including the raw BCD values read and the decoded time and date.

Definition at line 102 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.7.3.3 `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
------------------	-------------------------	---

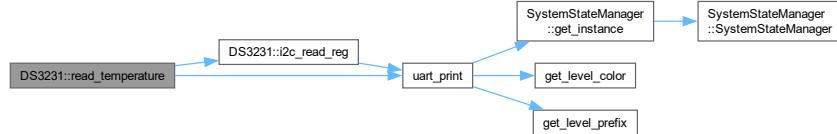
Returns

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

Here is the call graph for this function:

**7.7.3.4 set_unix_time()**

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

Sets the time using a Unix timestamp.

Sets the [DS3231](#) clock using a Unix timestamp.

Parameters

<code>in</code>	<code>unix_time</code>	Time as seconds since Unix epoch (1970-01-01 00:00:00 UTC)
-----------------	------------------------	--

Returns

0 on success, -1 on failure

Parameters

in	<i>unix_time</i>	The time in seconds since the Unix epoch (1970-01-01 00:00:00 UTC)
----	------------------	--

Returns

0 on success, -1 on failure

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

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

Here is the call graph for this function:



Here is the caller graph for this function:

**7.7.3.5 `get_unix_time()`**

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

Gets the current time as a Unix timestamp.

Gets the current time from [DS3231](#) as a Unix timestamp.

Returns

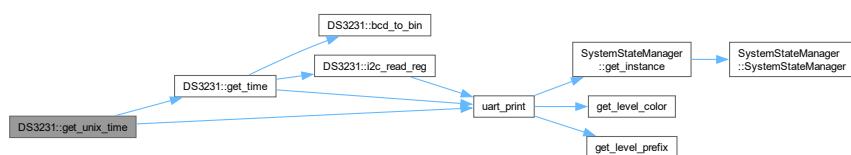
Unix timestamp, or -1 on error

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

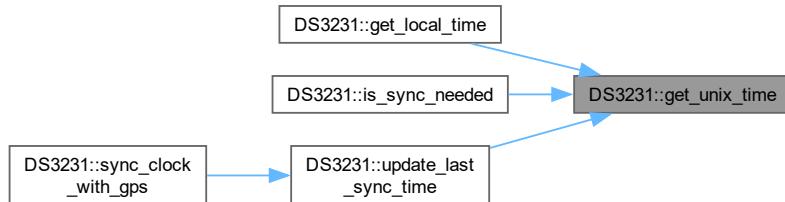
Reads the current time from the [DS3231](#) RTC and converts it to a Unix timestamp. This function properly handles the conversion between the internal [ds3231_data_t](#) format and the tm structure used by the C standard library.

Definition at line 202 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.7.3.6 `clock_enable()`**

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

Enables the [DS3231](#) clock oscillator.

Enables the [DS3231](#)'s oscillator.

Returns

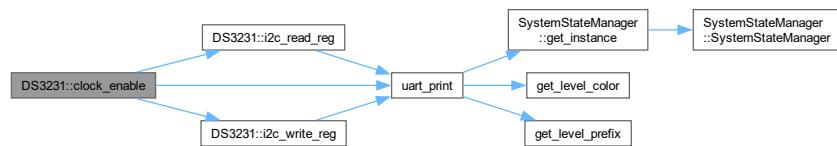
0 on success, -1 on failure

0 on success, -1 on failure

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

Definition at line 239 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.3.7 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 271 of file [DS3231.cpp](#).

7.7.3.8 set_timezone_offset()

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

Sets the timezone offset.

Parameters

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

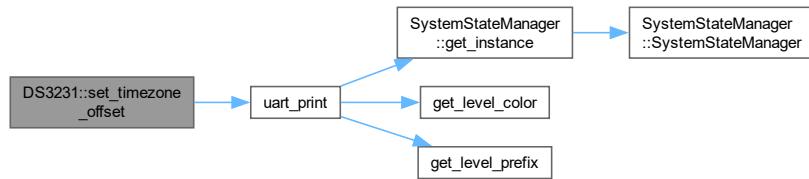
Sets the timezone offset in minutes relative to UTC. This value is used when converting between UTC and local time. The function validates that the offset is within a valid range (-720 to +720 minutes, which corresponds to -12 to +12 hours).

Note

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

Definition at line 286 of file [DS3231.cpp](#).

Here is the call graph for this function:

**7.7.3.9 get_clock_sync_interval()**

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

Gets the clock synchronization interval.

Gets the currently configured clock synchronization interval.

Returns

Sync interval in minutes

The sync interval in minutes

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

Definition at line 303 of file [DS3231.cpp](#).

7.7.3.10 set_clock_sync_interval()

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

Sets the clock synchronization interval.

Parameters

	<i>interval_minutes</i>	Interval in minutes (1-43200)
in	<i>interval_minutes</i>	The desired sync interval in minutes

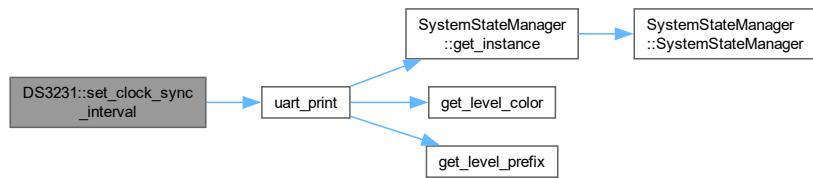
Sets how frequently the clock should be synchronized with an external time source (such as GPS). The function validates that the interval is within a valid range (1 minute to 43200 minutes, which is 30 days).

Note

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

Definition at line 317 of file [DS3231.cpp](#).

Here is the call graph for this function:



7.7.3.11 `get_last_sync_time()`

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

Gets the timestamp of the last clock synchronization.

Gets the timestamp of the last successful clock synchronization.

Returns

Unix timestamp of last sync, 0 if never synced

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

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

Definition at line 334 of file [DS3231.cpp](#).

7.7.3.12 update_last_sync_time()

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

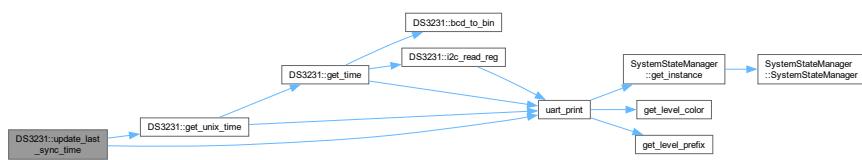
Updates the last sync time to current time.

Updates the last sync timestamp to the current time.

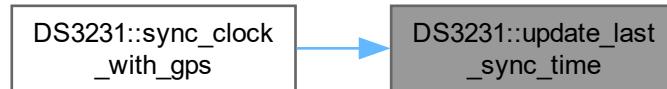
Records the current time as the last successful synchronization time. This should be called after successfully setting the time from an external source (such as GPS). The function logs the update with an informational message.

Definition at line 347 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.3.13 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.

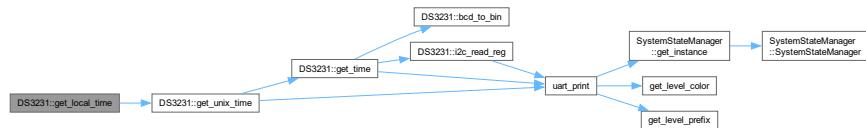
Returns

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

Here is the call graph for this function:

**7.7.3.14 is_sync_needed()**

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

Checks if clock synchronization is needed.

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

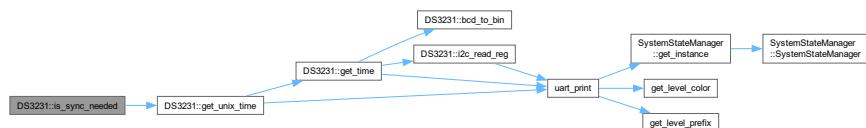
Returns

true if sync interval has elapsed since last sync, false otherwise
 true if synchronization is needed, false otherwise

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

Definition at line 379 of file [DS3231.cpp](#).

Here is the call graph for this function:

**7.7.3.15 sync_clock_with_gps()**

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

Synchronizes clock with GPS data.

Synchronizes the RTC with time from GPS.

Returns

true if sync successful, false otherwise

Parameters

in	<i>nmea_data</i>	Reference to NMEA data containing time information
----	------------------	--

Returns

true if synchronization succeeded, false if it failed

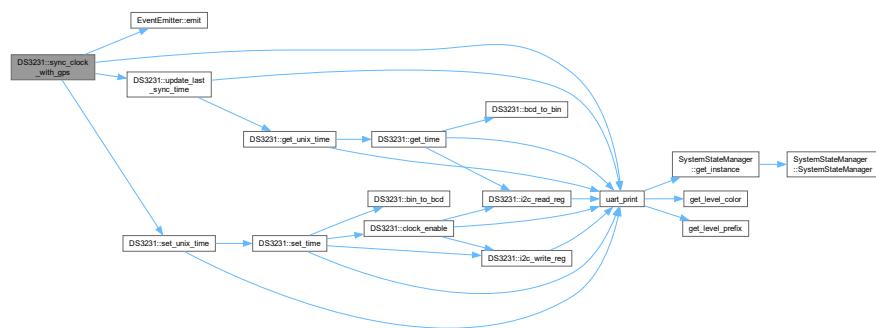
This method attempts to extract valid time data from the provided NMEA data and use it to update the RTC. It performs validity checks on the GPS data before attempting synchronization. If successful, it updates the last sync time and emits a SYNCED event. If unsuccessful, it emits a SYNC_FAILED event.

Note

This function emits events to the [EventEmitter](#) system that can be monitored by other components of the system.

Definition at line 409 of file [DS3231.cpp](#).

Here is the call graph for this function:

**7.7.3.16 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

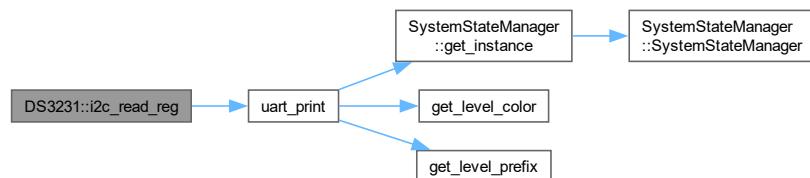
This method performs a thread-safe I²C read operation from the [DS3231](#). It first writes the register address to the device, then reads the requested number of bytes. All access is protected by a mutex to prevent concurrent I²C operations that could corrupt data.

Note

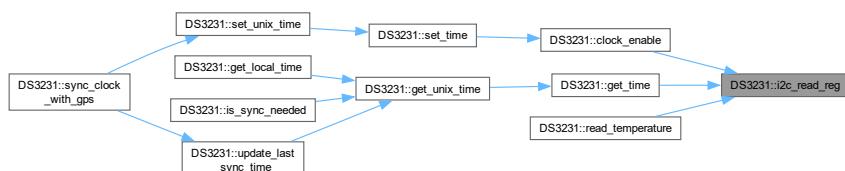
This is a low-level method used internally by the class.

Definition at line 455 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**7.7.3.17 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

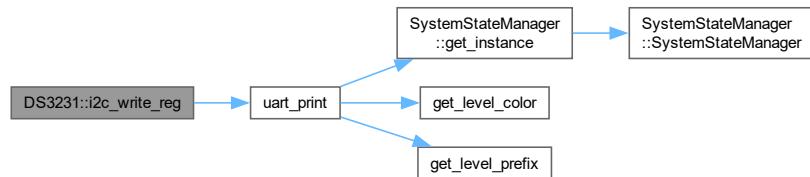
This method performs a thread-safe I²C write operation to the [DS3231](#). It combines the register address and data into a single buffer and sends it to the device. All access is protected by a mutex to prevent concurrent I²C operations that could corrupt data.

Note

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

Definition at line 496 of file [DS3231.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.3.18 bin_to_bcd()

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

Converts binary value to BCD (Binary Coded Decimal)

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

Parameters

in	<i>data</i>	Binary value to convert (0-99)
----	-------------	--------------------------------

Returns

BCD representation of the input value

Parameters

in	<i>data</i>	Binary value to convert (0-99)
----	-------------	--------------------------------

Returns

BCD representation of the input value

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

Definition at line [526](#) of file [DS3231.cpp](#).

Here is the caller graph for this function:



7.7.3.19 bcd_to_bin()

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

Converts BCD (Binary Coded Decimal) to binary value.

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

Parameters

<code>in</code>	<code>bcd</code>	BCD value to convert
-----------------	------------------	----------------------

Returns

Binary representation of the input BCD value

Parameters

<code>in</code>	<code>bcd</code>	BCD value to convert
-----------------	------------------	----------------------

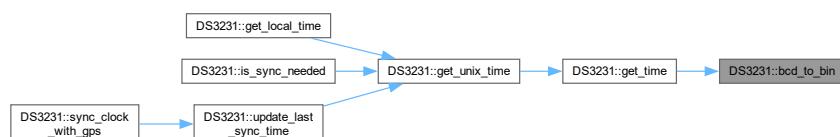
Returns

Binary representation of the input BCD value

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

Definition at line [542](#) of file [DS3231.cpp](#).

Here is the caller graph for this function:



7.7.4 Member Data Documentation

7.7.4.1 i2c

`i2c_inst_t* DS3231::i2c [private]`

Pointer to I2C instance.

Definition at line [226](#) of file [DS3231.h](#).

7.7.4.2 ds3231_addr

`uint8_t DS3231::ds3231_addr [private]`

I2C address of [DS3231](#).

Definition at line [227](#) of file [DS3231.h](#).

7.7.4.3 `clock_mutex_`

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

Mutex for thread-safe I2C access.

Definition at line 265 of file [DS3231.h](#).

7.7.4.4 `timezone_offset_minutes_`

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

Timezone offset in minutes, default: UTC.

Definition at line 266 of file [DS3231.h](#).

7.7.4.5 `sync_interval_minutes_`

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

Sync interval in minutes, default: 24 hours.

Definition at line 267 of file [DS3231.h](#).

7.7.4.6 `last_sync_time_`

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

Last sync timestamp, 0 = never synced.

Definition at line 268 of file [DS3231.h](#).

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

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

7.8 `ds3231_data_t` Struct Reference

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

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

Public Attributes

- `uint8_t seconds`
Seconds (0-59)
- `uint8_t minutes`
Minutes (0-59)
- `uint8_t hours`
Hours (0-23)
- `uint8_t day`
Day of the week (1-7)
- `uint8_t date`
Date (1-31)
- `uint8_t month`
Month (1-12)
- `uint8_t year`
Year (0-99)
- `bool century`
Century flag (0-1)

7.8.1 Detailed Description

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

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

7.8.2 Member Data Documentation

7.8.2.1 seconds

```
uint8_t ds3231_data_t::seconds
```

Seconds (0-59)

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

7.8.2.2 minutes

```
uint8_t ds3231_data_t::minutes
```

Minutes (0-59)

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

7.8.2.3 hours

```
uint8_t ds3231_data_t::hours
```

Hours (0-23)

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

7.8.2.4 day

```
uint8_t ds3231_data_t::day
```

Day of the week (1-7)

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

7.8.2.5 date

```
uint8_t ds3231_data_t::date
```

Date (1-31)

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

7.8.2.6 month

```
uint8_t ds3231_data_t::month
```

Month (1-12)

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

7.8.2.7 year

```
uint8_t ds3231_data_t::year
```

Year (0-99)

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

7.8.2.8 century

```
bool ds3231_data_t::century
```

Century flag (0-1)

Definition at line 98 of file [DS3231.h](#).

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

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

7.9 EventEmitter Class Reference

Provides a static method for emitting events.

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

Static Public Member Functions

- template<typename T>
static void [emit](#) ([EventGroup group](#), T event)
Emits an event.

7.9.1 Detailed Description

Provides a static method for emitting events.

Definition at line [319](#) of file [event_manager.h](#).

7.9.2 Member Function Documentation

7.9.2.1 [emit\(\)](#)

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

Emits an event.

Parameters

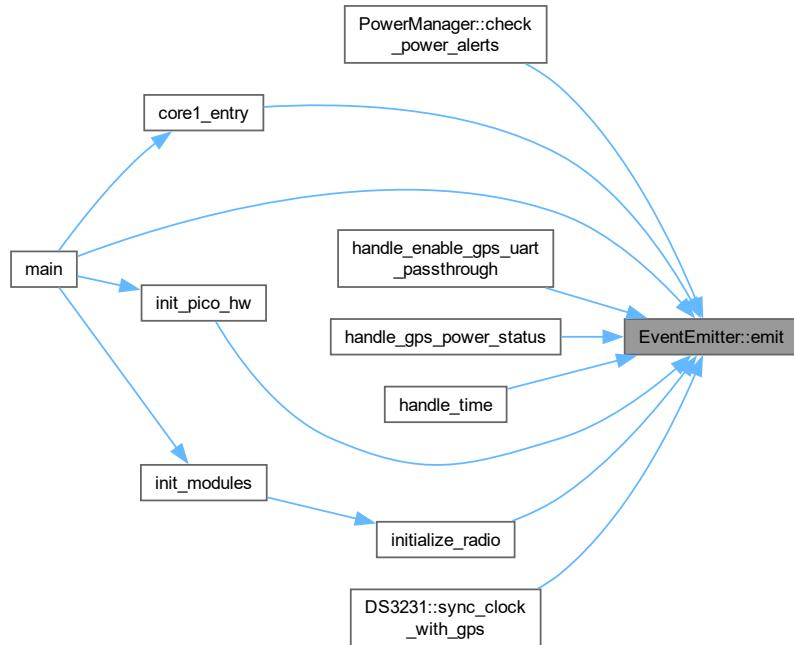
<i>group</i>	The event group.
<i>event</i>	The event identifier.

Template Parameters

<i>T</i>	The type of the event identifier.
----------	-----------------------------------

Definition at line [328](#) of file [event_manager.h](#).

Here is the caller graph for this function:



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

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

7.10 EventLog Class Reference

Represents a single event log entry.

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

Public Member Functions

- `std::string to_string () const`
Converts the `EventLog` to a string representation.

Public Attributes

- `uint16_t id`
Sequence number.
- `uint32_t timestamp`
Unix timestamp or system time.
- `uint8_t group`
Event group identifier.
- `uint8_t event`
Specific event identifier.

7.10.1 Detailed Description

Represents a single event log entry.

Definition at line 142 of file [event_manager.h](#).

7.10.2 Member Function Documentation

7.10.2.1 `to_string()`

```
std::string EventLog::to_string () const [inline]
```

Converts the [EventLog](#) to a string representation.

Returns

A string representation of the [EventLog](#).

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

Here is the caller graph for this function:



7.10.3 Member Data Documentation

7.10.3.1 `id`

```
uint16_t EventLog::id
```

Sequence number.

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

7.10.3.2 `timestamp`

```
uint32_t EventLog::timestamp
```

Unix timestamp or system time.

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

7.10.3.3 group

```
uint8_t EventLog::group
```

Event group identifier.

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

7.10.3.4 event

```
uint8_t EventLog::event
```

Specific event identifier.

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

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

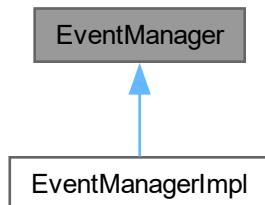
- lib/eventman/[event_manager.h](#)

7.11 EventManager Class Reference

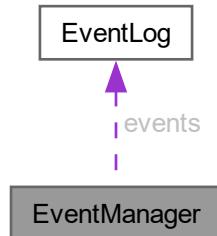
Manages the event logging system.

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

Inheritance diagram for EventManager:



Collaboration diagram for EventManager:



Public Member Functions

- `EventManager ()`
Constructor for the `EventManager`.
- `virtual ~EventManager ()=default`
Virtual destructor for the `EventManager`.
- `virtual void init ()`
Initializes the `EventManager`.
- `void log_event (uint8_t group, uint8_t event)`
Logs an event.
- `const EventLog & get_event (size_t index) const`
Retrieves an event from the event buffer.
- `size_t get_event_count () const`
Gets the number of events in the buffer.
- `virtual bool save_to_storage ()=0`
Saves the events to storage.
- `virtual bool load_from_storage ()=0`
Loads the events from storage.

Protected Attributes

- `EventLog events [EVENT_BUFFER_SIZE]`
Event buffer.
- `size_t eventCount`
Number of events in the buffer.
- `size_t writeIndex`
Index of the next event to be written.
- `mutex_t eventMutex`
Mutex for protecting the event buffer.
- `size_t eventsSinceFlush`
Number of events since last flush to storage.

Static Protected Attributes

- `static uint16_t nextEventId = 0`
Static event ID counter.

7.11.1 Detailed Description

Manages the event logging system.

Definition at line 171 of file `event_manager.h`.

7.11.2 Constructor & Destructor Documentation

7.11.2.1 EventManager()

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

Constructor for the `EventManager`.

Initializes the event buffer, mutex, and other internal variables.

Definition at line 177 of file `event_manager.h`.

7.11.2.2 ~EventManager()

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

Virtual destructor for the [EventManager](#).

7.11.3 Member Function Documentation

7.11.3.1 init()

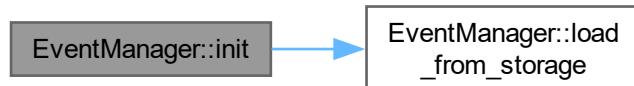
```
virtual void EventManager::init () [inline], [virtual]
```

Initializes the [EventManager](#).

Loads events from storage.

Definition at line 194 of file [event_manager.h](#).

Here is the call graph for this function:



Here is the caller graph for this function:



7.11.3.2 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 216 of file [event_manager.h](#).

7.11.3.3 `save_to_storage()`

```
virtual bool EventManager::save_to_storage () [pure virtual]
```

Saves the events to storage.

Returns

True if the events were successfully saved, false otherwise.

Implemented in [EventManagerImpl](#).

Here is the caller graph for this function:



7.11.3.4 `load_from_storage()`

```
virtual bool EventManager::load_from_storage () [pure virtual]
```

Loads the events from storage.

Returns

True if the events were successfully loaded, false otherwise.

Implemented in [EventManagerImpl](#).

Here is the caller graph for this function:



7.11.4 Member Data Documentation

7.11.4.1 `events`

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

Event buffer.

Definition at line 232 of file [event_manager.h](#).

7.11.4.2 eventCount

```
size_t EventManager::eventCount [protected]
```

Number of events in the buffer.

Definition at line 234 of file [event_manager.h](#).

7.11.4.3 writeIndex

```
size_t EventManager::writeIndex [protected]
```

Index of the next event to be written.

Definition at line 236 of file [event_manager.h](#).

7.11.4.4 eventMutex

```
mutex_t EventManager::eventMutex [protected]
```

Mutex for protecting the event buffer.

Definition at line 238 of file [event_manager.h](#).

7.11.4.5 nextEventId

```
uint16_t EventManager::nextEventId = 0 [static], [protected]
```

Static event ID counter.

Definition at line 240 of file [event_manager.h](#).

7.11.4.6 eventsSinceFlush

```
size_t EventManager::eventsSinceFlush [protected]
```

Number of events since last flush to storage.

Definition at line 242 of file [event_manager.h](#).

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

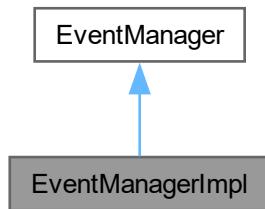
- lib/eventman/[event_manager.h](#)
- lib/eventman/[event_manager.cpp](#)

7.12 EventManagerImpl Class Reference

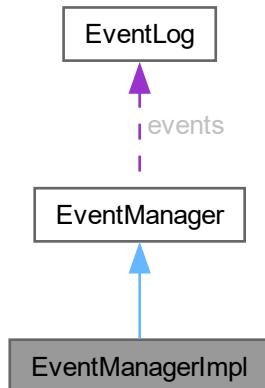
Implementation of the [EventManager](#) class.

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

Inheritance diagram for EventManagerImpl:



Collaboration diagram for EventManagerImpl:



Public Member Functions

- [EventManagerImpl \(\)](#)
Constructor for the EventManagerImpl.
- [bool save_to_storage \(\) override](#)
Saves the events to storage.
- [bool load_from_storage \(\) override](#)
Loads the events from storage.

Public Member Functions inherited from [EventManager](#)

- [EventManager \(\)](#)
Constructor for the [EventManager](#).
- [virtual ~EventManager \(\)=default](#)
Virtual destructor for the [EventManager](#).
- [virtual void init \(\)](#)
Initializes the [EventManager](#).
- [void log_event \(uint8_t group, uint8_t event\)](#)
Logs an event.
- [const EventLog & get_event \(size_t index\) const](#)
Retrieves an event from the event buffer.
- [size_t get_event_count \(\) const](#)
Gets the number of events in the buffer.

Additional Inherited Members

Protected Attributes inherited from [EventManager](#)

- [EventLog events \[EVENT_BUFFER_SIZE\]](#)
Event buffer.
- [size_t eventCount](#)
Number of events in the buffer.
- [size_t writeIndex](#)
Index of the next event to be written.
- [mutex_t eventMutex](#)
Mutex for protecting the event buffer.
- [size_t eventsSinceFlush](#)
Number of events since last flush to storage.

Static Protected Attributes inherited from [EventManager](#)

- [static uint16_t nextEventId = 0](#)
Static event ID counter.

7.12.1 Detailed Description

Implementation of the [EventManager](#) class.

Definition at line 250 of file [event_manager.h](#).

7.12.2 Constructor & Destructor Documentation

7.12.2.1 EventManagerImpl()

```
EventManagerImpl::EventManagerImpl () [inline]
```

Constructor for the [EventManagerImpl](#).

Initializes the [EventManagerImpl](#) and calls the init method.

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

Here is the call graph for this function:



7.12.3 Member Function Documentation

7.12.3.1 save_to_storage()

```
bool EventManagerImpl::save_to_storage () [inline], [override], [virtual]
```

Saves the events to storage.

Returns

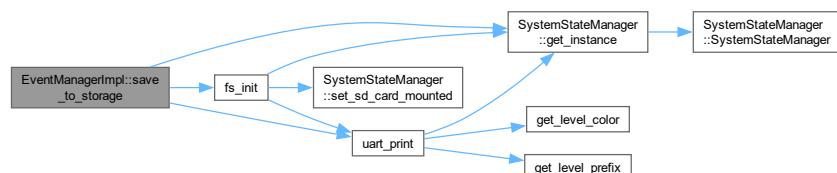
True if the events were successfully saved, false otherwise.

This method is not yet implemented.

Implements [EventManager](#).

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

Here is the call graph for this function:



7.12.3.2 `load_from_storage()`

```
bool EventManagerImpl::load_from_storage () [inline], [override], [virtual]
```

Loads the events from storage.

Returns

True if the events were successfully loaded, false otherwise.

This method is not yet implemented.

Implements [EventManager](#).

Definition at line 303 of file [event_manager.h](#).

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

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

7.13 FileHandle Struct Reference

```
#include <storage.h>
```

Public Attributes

- int [fd](#)
- bool [is_open](#)

7.13.1 Detailed Description

Definition at line 19 of file [storage.h](#).

7.13.2 Member Data Documentation

7.13.2.1 `fd`

```
int FileHandle::fd
```

Definition at line 20 of file [storage.h](#).

7.13.2.2 `is_open`

```
bool FileHandle::is_open
```

Definition at line 21 of file [storage.h](#).

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

- lib/storage/[storage.h](#)

7.14 Frame Struct Reference

Represents a communication frame used for data exchange.

```
#include <protocol.h>
```

Public Attributes

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

7.14.1 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.

Example Usage:

```
// Creating a Frame instance
Frame myFrame;
myFrame.header = FRAME_BEGIN;
myFrame.direction = 1;
myFrame.operationType = OperationType::ANS;
myFrame.group = 2;
myFrame.command = 5;
myFrame.value = "25.5";
myFrame.unit = "VOLT";
myFrame.footer = FRAME_END;

// Encoding the Frame to a string
std::string encodedFrame = frame_encode(myFrame);
```

Example Instances:

```
// Example of a GET command
Frame getCommand;
getCommand.header = FRAME_BEGIN;
getCommand.direction = 0;
getCommand.operationType = OperationType::GET;
getCommand.group = 1;
getCommand.command = 10;
```

```

getCommand.value = "";
getCommand.unit = "";
getCommand.footer = FRAME_END;

// Example of an ANSWER command
Frame answerCommand;
answerCommand.header = FRAME_BEGIN;
answerCommand.direction = 1;
answerCommand.operationType = OperationType::ANS;
answerCommand.group = 1;
answerCommand.command = 10;
answerCommand.value = "OK";
answerCommand.unit = "";
answerCommand.footer = FRAME_END;

```

Example of Encoded Frames:

```

// Encoded GET command example:
// KBST;0;GET;1;10;;TSBK

// Encoded SET command example:
// KBST;0;SET;2;5;25.5;VOLT;TSBK

// Encoded ANSWER command example:
// KBST;1;ANS;1;10;OK;;TSBK

// Encoded ERROR command example:
// KBST;1;ERR;3;1;Invalid Parameter;;TSBK

// Encoded INFO command example:
// KBST;1;INF;4;2;System Booted;;TSBK

```

Definition at line 227 of file [protocol.h](#).

7.14.2 Member Data Documentation

7.14.2.1 header

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

Definition at line 228 of file [protocol.h](#).

7.14.2.2 direction

```
uint8_t Frame::direction
```

Definition at line 229 of file [protocol.h](#).

7.14.2.3 operationType

```
OperationType Frame::operationType
```

Definition at line 230 of file [protocol.h](#).

7.14.2.4 group

```
uint8_t Frame::group
```

Definition at line 231 of file [protocol.h](#).

7.14.2.5 command

```
uint8_t Frame::command
```

Definition at line 232 of file [protocol.h](#).

7.14.2.6 value

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

Definition at line 233 of file [protocol.h](#).

7.14.2.7 unit

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

Definition at line 234 of file [protocol.h](#).

7.14.2.8 footer

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

Definition at line 235 of file [protocol.h](#).

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

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

7.15 HMC5883L Class Reference

```
#include <HMC5883L.h>
```

Public Member Functions

- [HMC5883L](#) (*i2c_inst_t* **i2c*, *uint8_t* **address**=0x0D)
- bool [init](#) ()
- bool [read](#) (*int16_t* &*x*, *int16_t* &*y*, *int16_t* &*z*)

Private Member Functions

- bool [write_register](#) (*uint8_t* *reg*, *uint8_t* *value*)
- bool [read_register](#) (*uint8_t* *reg*, *uint8_t* **buffer*, *size_t* *length*)

Private Attributes

- i2c_inst_t * [i2c](#)
- uint8_t [address](#)

7.15.1 Detailed Description

Definition at line [6](#) of file [HMC5883L.h](#).

7.15.2 Constructor & Destructor Documentation

7.15.2.1 HMC5883L()

```
HMC5883L::HMC5883L (
    i2c_inst_t * i2c,
    uint8_t address = 0x0D)
```

Definition at line [3](#) of file [HMC5883L.cpp](#).

7.15.3 Member Function Documentation

7.15.3.1 init()

```
bool HMC5883L::init ()
```

Definition at line [5](#) of file [HMC5883L.cpp](#).

Here is the call graph for this function:



7.15.3.2 `read()`

```
bool HMC5883L::read (
    int16_t & x,
    int16_t & y,
    int16_t & z)
```

Definition at line 13 of file [HMC5883L.cpp](#).

Here is the call graph for this function:



7.15.3.3 `write_register()`

```
bool HMC5883L::write_register (
    uint8_t reg,
    uint8_t value) [private]
```

Definition at line 28 of file [HMC5883L.cpp](#).

Here is the caller graph for this function:



7.15.3.4 `read_register()`

```
bool HMC5883L::read_register (
    uint8_t reg,
    uint8_t * buffer,
    size_t length) [private]
```

Definition at line 33 of file [HMC5883L.cpp](#).

Here is the caller graph for this function:



7.15.4 Member Data Documentation

7.15.4.1 i2c

```
i2c_inst_t* HMC5883L::i2c [private]
```

Definition at line 13 of file [HMC5883L.h](#).

7.15.4.2 address

```
uint8_t HMC5883L::address [private]
```

Definition at line 14 of file [HMC5883L.h](#).

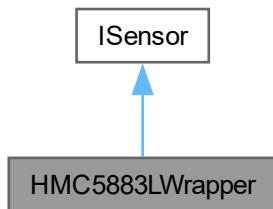
The documentation for this class was generated from the following files:

- lib/sensors/HMC5883L/[HMC5883L.h](#)
- lib/sensors/HMC5883L/[HMC5883L.cpp](#)

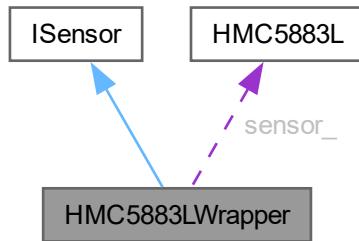
7.16 HMC5883LWrapper Class Reference

```
#include <HMC5883L_WRAPPER.h>
```

Inheritance diagram for HMC5883LWrapper:



Collaboration diagram for HMC5883LWrapper:



Public Member Functions

- `HMC5883LWrapper (i2c_inst_t *i2c)`
- `bool init () override`
- `float read_data (SensorDataTypelIdentifier type) override`
- `bool is_initialized () const override`
- `SensorType get_type () const override`
- `bool configure (const std::map< std::string, std::string > &config) override`
- `uint8_t get_address () const override`

Public Member Functions inherited from [ISensor](#)

- `virtual ~ISensor ()=default`

Private Attributes

- `HMC5883L sensor_`
- `bool initialized_`

7.16.1 Detailed Description

Definition at line 7 of file [HMC5883L_WRAPPER.h](#).

7.16.2 Constructor & Destructor Documentation

7.16.2.1 `HMC5883LWrapper()`

```
HMC5883LWrapper::HMC5883LWrapper (
    i2c_inst_t * i2c)
```

Definition at line 5 of file [HMC5883L_WRAPPER.cpp](#).

7.16.3 Member Function Documentation

7.16.3.1 init()

```
bool HMC5883LWrapper::init () [override], [virtual]
```

Implements [ISensor](#).

Definition at line [7](#) of file [HMC5883L_WRAPPER.cpp](#).

7.16.3.2 read_data()

```
float HMC5883LWrapper::read_data (
    SensorDataTypeIdentifier type) [override], [virtual]
```

Implements [ISensor](#).

Definition at line [12](#) of file [HMC5883L_WRAPPER.cpp](#).

7.16.3.3 is_initialized()

```
bool HMC5883LWrapper::is_initialized () const [override], [virtual]
```

Implements [ISensor](#).

Definition at line [35](#) of file [HMC5883L_WRAPPER.cpp](#).

7.16.3.4 get_type()

```
SensorType HMC5883LWrapper::get_type () const [override], [virtual]
```

Implements [ISensor](#).

Definition at line [39](#) of file [HMC5883L_WRAPPER.cpp](#).

7.16.3.5 configure()

```
bool HMC5883LWrapper::configure (
    const std::map< std::string, std::string > & config) [override], [virtual]
```

Implements [ISensor](#).

Definition at line [43](#) of file [HMC5883L_WRAPPER.cpp](#).

7.16.3.6 get_address()

```
uint8_t HMC5883LWrapper::get_address () const [inline], [override], [virtual]
```

Implements [ISensor](#).

Definition at line [16](#) of file [HMC5883L_WRAPPER.h](#).

7.16.4 Member Data Documentation

7.16.4.1 `sensor_`

[HMC5883L](#) HMC5883LWrapper::sensor_ [private]

Definition at line 21 of file [HMC5883L_WRAPPER.h](#).

7.16.4.2 `initialized_`

bool HMC5883LWrapper::initialized_ [private]

Definition at line 22 of file [HMC5883L_WRAPPER.h](#).

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

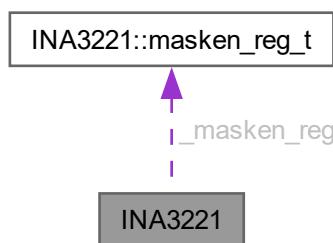
- lib/sensors/HMC5883L/[HMC5883L_WRAPPER.h](#)
- lib/sensors/HMC5883L/[HMC5883L_WRAPPER.cpp](#)

7.17 INA3221 Class Reference

[INA3221](#) Triple-Channel Power Monitor driver class.

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

Collaboration diagram for INA3221:



Classes

- struct [conf_reg_t](#)
Configuration register bit fields.
- struct [masken_reg_t](#)
Mask/Enable register bit fields.

Public Member Functions

- `INA3221 (ina3221_addr_t addr, i2c_inst_t *i2c)`
`Constructor for INA3221 class.`
- `bool begin ()`
`Initialize the INA3221 device.`
- `uint16_t read_register (ina3221_reg_t reg)`
`Read a register from the device.`
- `void reset ()`
`Reset the INA3221 to default settings.`
- `void set_mode_power_down ()`
`Set device to power-down mode.`
- `void set_mode_continuous ()`
`Set device to continuous measurement mode.`
- `void set_mode_triggered ()`
`Set device to triggered measurement mode.`
- `void set_shunt_measurement_enable ()`
`Enable shunt voltage measurements.`
- `void set_shunt_measurement_disable ()`
`Disable shunt voltage measurements.`
- `void set_bus_measurement_enable ()`
`Enable bus voltage measurements.`
- `void set_bus_measurement_disable ()`
`Disable bus voltage measurements.`
- `void set_averaging_mode (ina3221_avg_mode_t mode)`
`Set the averaging mode for measurements.`
- `void set_bus_conversion_time (ina3221_conv_time_t convTime)`
`Set bus voltage conversion time.`
- `void set_shunt_conversion_time (ina3221_conv_time_t convTime)`
`Set shunt voltage conversion time.`
- `uint16_t get_manufacturer_id ()`
`Get the manufacturer ID of the device.`
- `uint16_t get_die_id ()`
`Get the die ID of the device.`
- `int32_t get_shunt_voltage (ina3221_ch_t channel)`
`Get shunt voltage for a specific channel.`
- `float get_current (ina3221_ch_t channel)`
- `float get_current_ma (ina3221_ch_t channel)`
`Get current for a specific channel.`
- `float get_voltage (ina3221_ch_t channel)`
`Get bus voltage for a specific channel.`
- `void set_warn_alert_limit (ina3221_ch_t channel, float voltage_v)`
`Set warning alert voltage threshold for a channel.`
- `void set_crit_alert_limit (ina3221_ch_t channel, float voltage_v)`
`Set critical alert voltage threshold for a channel.`
- `void set_power_valid_limit (float voltage_upper_v, float voltage_lower_v)`
`Set power valid voltage range.`
- `void enable_alerts ()`
`Enable all alert functions.`
- `bool get_warn_alert (ina3221_ch_t channel)`
`Get warning alert status for a channel.`

- bool `get_crit_alert (ina3221_ch_t channel)`
Get critical alert status for a channel.
- bool `get_power_valid_alert ()`
Get power valid alert status.
- void `set_alert_latch (bool enable)`
Set alert latch mode.

Private Member Functions

- void `_read (ina3221_reg_t reg, uint16_t *val)`
Read a 16-bit register from the device.
- void `_write (ina3221_reg_t reg, uint16_t *val)`
Write a 16-bit value to a register.

Private Attributes

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

7.17.1 Detailed Description

[INA3221](#) Triple-Channel Power Monitor driver class.

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

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

7.17.2 Member Function Documentation

7.17.2.1 `_read()`

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

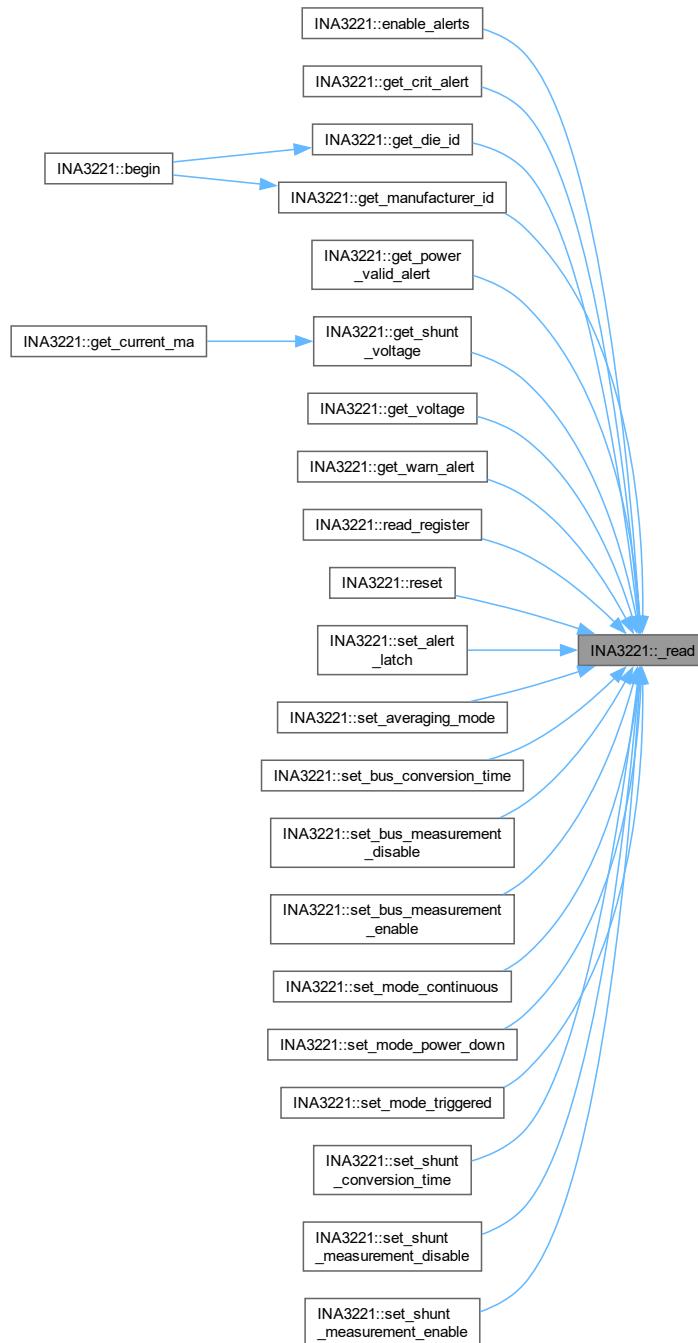
Read a 16-bit register from the device.

Parameters

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

Definition at line 513 of file [INA3221.cpp](#).

Here is the caller graph for this function:



7.17.2.2 `_write()`

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

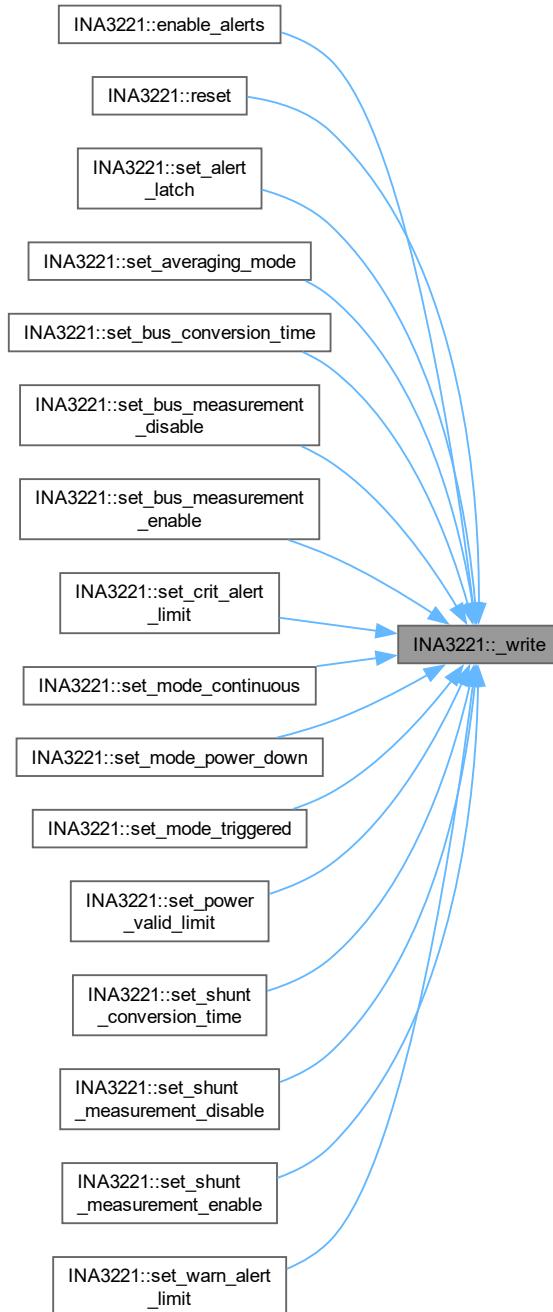
Write a 16-bit value to a register.

Parameters

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

Definition at line 539 of file [INA3221.cpp](#).

Here is the caller graph for this function:



7.17.2.3 `get_current()`

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

7.17.3 Member Data Documentation

7.17.3.1 _i2c_addr

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

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

7.17.3.2 _i2c

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

Definition at line 138 of file [INA3221.h](#).

7.17.3.3 _shuntRes

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

Definition at line 141 of file [INA3221.h](#).

7.17.3.4 _filterRes

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

Definition at line 144 of file [INA3221.h](#).

7.17.3.5 _masken_reg

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

Definition at line 147 of file [INA3221.h](#).

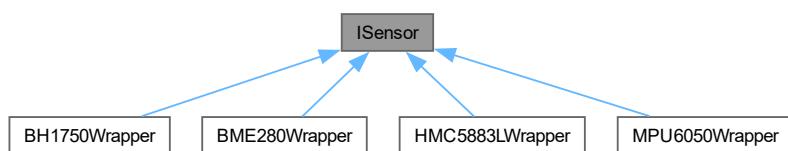
The documentation for this class was generated from the following files:

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

7.18 ISensor Class Reference

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

Inheritance diagram for ISensor:



Public Member Functions

- virtual `~ISensor()`=default
- virtual bool `init()`=0
- virtual float `read_data(SensorDataTypelIdentifier type)`=0
- virtual bool `is_initialized()` const =0
- virtual `SensorType get_type()` const =0
- virtual bool `configure(const std::map<std::string, std::string> &config)`=0
- virtual uint8_t `get_address()` const =0

7.18.1 Detailed Description

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

7.18.2 Constructor & Destructor Documentation

7.18.2.1 `~ISensor()`

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

7.18.3 Member Function Documentation

7.18.3.1 `init()`

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

Implemented in [BH1750Wrapper](#), [BME280Wrapper](#), [HMC5883LWrapper](#), and [MPU6050Wrapper](#).

7.18.3.2 `read_data()`

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

Implemented in [BH1750Wrapper](#), [BME280Wrapper](#), [HMC5883LWrapper](#), and [MPU6050Wrapper](#).

7.18.3.3 `is_initialized()`

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

Implemented in [BH1750Wrapper](#), [BME280Wrapper](#), [HMC5883LWrapper](#), and [MPU6050Wrapper](#).

7.18.3.4 `get_type()`

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

Implemented in [BH1750Wrapper](#), [BME280Wrapper](#), [HMC5883LWrapper](#), and [MPU6050Wrapper](#).

7.18.3.5 configure()

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

Implemented in [BH1750Wrapper](#), [BME280Wrapper](#), [HMC5883LWrapper](#), and [MPU6050Wrapper](#).

7.18.3.6 get_address()

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

Implemented in [BH1750Wrapper](#), [BME280Wrapper](#), and [HMC5883LWrapper](#).

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

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

7.19 INA3221::masken_reg_t Struct Reference

Mask/Enable register bit fields.

Public Attributes

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

7.19.1 Detailed Description

Mask/Enable register bit fields.

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

7.19.2 Member Data Documentation

7.19.2.1 conv_ready

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

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

7.19.2.2 timing_ctrl_alert

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

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

7.19.2.3 pwr_valid_alert

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

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

7.19.2.4 warn_alert_ch3

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

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

7.19.2.5 warn_alert_ch2

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

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

7.19.2.6 warn_alert_ch1

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

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

7.19.2.7 shunt_sum_alert

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

Definition at line 124 of file [INA3221.h](#).

7.19.2.8 crit_alert_ch3

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

Definition at line 125 of file [INA3221.h](#).

7.19.2.9 crit_alert_ch2

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

Definition at line 126 of file [INA3221.h](#).

7.19.2.10 crit_alert_ch1

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

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

7.19.2.11 crit_alert_latch_en

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

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

7.19.2.12 warn_alert_latch_en

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

Definition at line 129 of file [INA3221.h](#).

7.19.2.13 shunt_sum_en_ch3

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

Definition at line 130 of file [INA3221.h](#).

7.19.2.14 shunt_sum_en_ch2

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

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

7.19.2.15 shunt_sum_en_ch1

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

Definition at line 132 of file [INA3221.h](#).

7.19.2.16 reserved

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

Definition at line 133 of file [INA3221.h](#).

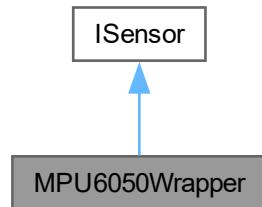
The documentation for this struct was generated from the following file:

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

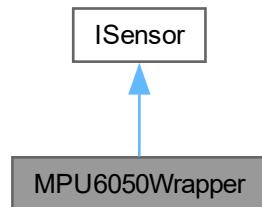
7.20 MPU6050Wrapper Class Reference

```
#include <MPU6050_WRAPPER.h>
```

Inheritance diagram for MPU6050Wrapper:



Collaboration diagram for MPU6050Wrapper:



Public Member Functions

- [MPU6050Wrapper \(\)](#)
- bool [init \(\)](#) override
- float [read_data \(SensorDataTypelIdentifier type\)](#) override
- bool [is_initialized \(\)](#) const override
- [SensorType get_type \(\)](#) const override
- bool [configure \(const std::map< std::string, std::string > &config\)](#)

Public Member Functions inherited from [ISensor](#)

- virtual `~ISensor ()=default`
- virtual `uint8_t get_address () const =0`

Private Attributes

- `MPU6050 sensor_`
- `bool initialized_ = false`

7.20.1 Detailed Description

Definition at line 9 of file [MPU6050_WRAPPER.h](#).

7.20.2 Constructor & Destructor Documentation

7.20.2.1 `MPU6050Wrapper()`

```
MPU6050Wrapper::MPU6050Wrapper ()
```

7.20.3 Member Function Documentation

7.20.3.1 `init()`

```
bool MPU6050Wrapper::init () [override], [virtual]
```

Implements [ISensor](#).

7.20.3.2 `read_data()`

```
float MPU6050Wrapper::read_data (
    SensorDataTypeIdentifier type) [override], [virtual]
```

Implements [ISensor](#).

7.20.3.3 `is_initialized()`

```
bool MPU6050Wrapper::is_initialized () const [override], [virtual]
```

Implements [ISensor](#).

7.20.3.4 `get_type()`

```
SensorType MPU6050Wrapper::get_type () const [override], [virtual]
```

Implements [ISensor](#).

7.20.3.5 `configure()`

```
bool MPU6050Wrapper::configure (
    const std::map< std::string, std::string > & config) [virtual]
```

Implements [ISensor](#).

7.20.4 Member Data Documentation

7.20.4.1 `sensor_`

```
MPU6050 MPU6050Wrapper::sensor_ [private]
```

Definition at line 11 of file [MPU6050_WRAPPER.h](#).

7.20.4.2 `initialized_`

```
bool MPU6050Wrapper::initialized_ = false [private]
```

Definition at line 12 of file [MPU6050_WRAPPER.h](#).

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

- lib/sensors/MPU6050/[MPU6050_WRAPPER.h](#)

7.21 NMEAData Class Reference

```
#include <NMEA_data.h>
```

Public Member Functions

- [NMEAData \(\)](#)
- void [update_rmc_tokens](#) (const std::vector< std::string > &tokens)
- void [update_gga_tokens](#) (const std::vector< std::string > &tokens)
- std::vector< std::string > [get_rmc_tokens](#) () const
- std::vector< std::string > [get_gga_tokens](#) () const
- bool [has_valid_time](#) () const
- time_t [get_unix_time](#) () const

Private Attributes

- std::vector< std::string > [rmc_tokens_](#)
- std::vector< std::string > [gga_tokens_](#)
- mutex_t [rmc_mutex_](#)
- mutex_t [gga_mutex_](#)

7.21.1 Detailed Description

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

7.21.2 Constructor & Destructor Documentation

7.21.2.1 NMEAData()

```
NMEAData::NMEAData ()
```

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

7.21.3 Member Function Documentation

7.21.3.1 update_rmc_tokens()

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

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

7.21.3.2 update_gga_tokens()

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

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

7.21.3.3 get_rmc_tokens()

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

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

7.21.3.4 get_gga_tokens()

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

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

7.21.3.5 has_valid_time()

```
bool NMEAData::has_valid_time () const
```

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

Here is the caller graph for this function:



7.21.3.6 get_unix_time()

```
time_t NMEAData::get_unix_time () const
```

Definition at line 40 of file [NMEA_data.cpp](#).

Here is the call graph for this function:



7.21.4 Member Data Documentation

7.21.4.1 rmc_tokens_

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

Definition at line 24 of file [NMEA_data.h](#).

7.21.4.2 gga_tokens_

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

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

7.21.4.3 rmc_mutex_

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

Definition at line 26 of file [NMEA_data.h](#).

7.21.4.4 gga_mutex_

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

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

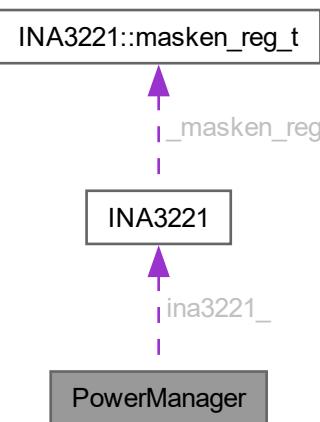
The documentation for this class was generated from the following files:

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

7.22 PowerManager Class Reference

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

Collaboration diagram for PowerManager:



Public Member Functions

- `PowerManager (i2c_inst_t *i2c)`
- `bool initialize ()`
- `std::string read_device_ids ()`
- `float get_current_charge_solar ()`
- `float get_current_charge_usb ()`
- `float get_current_charge_total ()`
- `float get_current_draw ()`
- `float get_voltage_battery ()`
- `float get_voltage_5v ()`
- `void configure (const std::map< std::string, std::string > &config)`
- `bool is_charging_solar ()`
- `bool is_charging_usb ()`
- `bool check_power_alerts ()`

Static Public Attributes

- `static constexpr float SOLAR_CURRENT_THRESHOLD = 50.0f`
- `static constexpr float USB_CURRENT_THRESHOLD = 50.0f`
- `static constexpr float VOLTAGE_LOW_THRESHOLD = 4.6f`
- `static constexpr float VOLTAGE_OVERCHARGE_THRESHOLD = 5.3f`
- `static constexpr float FALL_RATE_THRESHOLD = -0.02f`
- `static constexpr int FALLING_TREND_REQUIRED = 3`

Private Attributes

- `INA3221 ina3221_`
- `bool initialized_`
- `recursive_mutex_t powerman_mutex_`
- `bool charging_solar_active_ = false`
- `bool charging_usb_active_ = false`

7.22.1 Detailed Description

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

7.22.2 Constructor & Destructor Documentation

7.22.2.1 PowerManager()

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

Definition at line 6 of file [PowerManager.cpp](#).

7.22.3 Member Function Documentation

7.22.3.1 initialize()

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

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

7.22.3.2 read_device_ids()

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

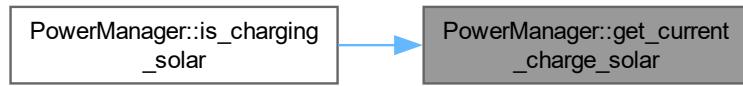
Definition at line 28 of file [PowerManager.cpp](#).

7.22.3.3 get_current_charge_solar()

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

Definition at line 74 of file [PowerManager.cpp](#).

Here is the caller graph for this function:



7.22.3.4 get_current_charge_usb()

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

Definition at line 58 of file [PowerManager.cpp](#).

Here is the caller graph for this function:



7.22.3.5 `get_current_charge_total()`

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

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

7.22.3.6 `get_current_draw()`

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

Definition at line [66](#) of file [PowerManager.cpp](#).

7.22.3.7 `get_voltage_battery()`

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

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

7.22.3.8 `get_voltage_5v()`

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

Definition at line [50](#) of file [PowerManager.cpp](#).

7.22.3.9 `configure()`

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

Definition at line [90](#) of file [PowerManager.cpp](#).

7.22.3.10 `is_charging_solar()`

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

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

Here is the call graph for this function:



7.22.3.11 `is_charging_usb()`

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

Definition at line 127 of file [PowerManager.cpp](#).

Here is the call graph for this function:



7.22.3.12 `check_power_alerts()`

```
bool PowerManager::check_power_alerts ()
```

Definition at line 135 of file [PowerManager.cpp](#).

Here is the call graph for this function:



7.22.4 Member Data Documentation

7.22.4.1 `SOLAR_CURRENT_THRESHOLD`

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

Definition at line 28 of file [PowerManager.h](#).

7.22.4.2 `USB_CURRENT_THRESHOLD`

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

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

7.22.4.3 VOLTAGE_LOW_THRESHOLD

```
float PowerManager::VOLTAGE_LOW_THRESHOLD = 4.6f [static], [constexpr]
```

Definition at line 30 of file [PowerManager.h](#).

7.22.4.4 VOLTAGE_OVERCHARGE_THRESHOLD

```
float PowerManager::VOLTAGE_OVERCHARGE_THRESHOLD = 5.3f [static], [constexpr]
```

Definition at line 31 of file [PowerManager.h](#).

7.22.4.5 FALL_RATE_THRESHOLD

```
float PowerManager::FALL_RATE_THRESHOLD = -0.02f [static], [constexpr]
```

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

7.22.4.6 FALLING_TREND_REQUIRED

```
int PowerManager::FALLING_TREND_REQUIRED = 3 [static], [constexpr]
```

Definition at line 33 of file [PowerManager.h](#).

7.22.4.7 ina3221_

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

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

7.22.4.8 initialized_

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

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

7.22.4.9 powerman_mutex_

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

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

7.22.4.10 charging_solar_active_

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

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

7.22.4.11 charging_usb_active_

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

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

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

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

7.23 SensorDataRecord Struct Reference

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

Public Member Functions

- `std::string to_csv () const`

Public Attributes

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

7.23.1 Detailed Description

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

7.23.2 Member Function Documentation

7.23.2.1 to_csv()

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

Definition at line 106 of file [telemetry_manager.h](#).

Here is the caller graph for this function:



7.23.3 Member Data Documentation

7.23.3.1 timestamp

```
uint32_t SensorDataRecord::timestamp
```

Unix timestamp of the record

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

7.23.3.2 temperature

```
float SensorDataRecord::temperature
```

Temperature in degrees Celsius

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

7.23.3.3 pressure

```
float SensorDataRecord::pressure
```

Pressure in hPa

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

7.23.3.4 humidity

```
float SensorDataRecord::humidity
```

Relative humidity in %

Definition at line 102 of file [telemetry_manager.h](#).

7.23.3.5 light

```
float SensorDataRecord::light
```

Light intensity in lux

Definition at line 103 of file [telemetry_manager.h](#).

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

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

7.24 SensorWrapper Class Reference

Manages different sensor types and provides a unified interface for accessing sensor data.

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

Public Member Functions

- bool `sensor_init (SensorType type, i2c_inst_t *i2c=nullptr)`
Initializes a given sensor type on the specified I2C bus.
- bool `sensor_configure (SensorType type, const std::map< std::string, std::string > &config)`
Configures an already initialized sensor with supplied settings.
- float `sensor_read_data (SensorType sensorType, SensorDataTypelIdentifier dataType)`
Reads a specific data type (e.g., temperature, humidity) from a sensor.
- `ISensor * get_sensor (SensorType type)`
- `std::vector< std::pair< SensorType, uint8_t > > scan_connected_sensors (i2c_inst_t *i2c)`
Scans the I2C bus for connected sensors.
- `std::vector< std::pair< SensorType, uint8_t > > get_available_sensors ()`
Retrieves a list of available sensor types with their addresses.

Static Public Member Functions

- static `SensorWrapper & get_instance ()`
Provides a global instance of SensorWrapper.

Private Member Functions

- `SensorWrapper ()`
Default constructor for SensorWrapper.

Private Attributes

- `std::map< SensorType, ISensor * > sensors`

7.24.1 Detailed Description

Manages different sensor types and provides a unified interface for accessing sensor data.

Definition at line 45 of file `ISensor.h`.

7.24.2 Constructor & Destructor Documentation

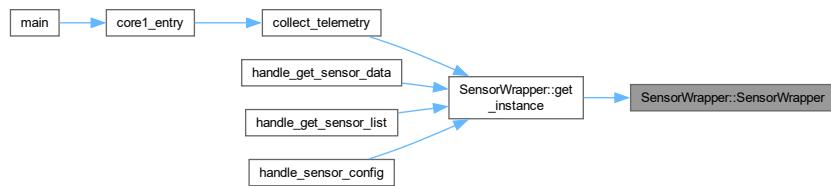
7.24.2.1 SensorWrapper()

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

Default constructor for [SensorWrapper](#).

Definition at line 98 of file [test_mocks.cpp](#).

Here is the caller graph for this function:



7.24.3 Member Function Documentation

7.24.3.1 get_instance()

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

Provides a global instance of [SensorWrapper](#).

Returns

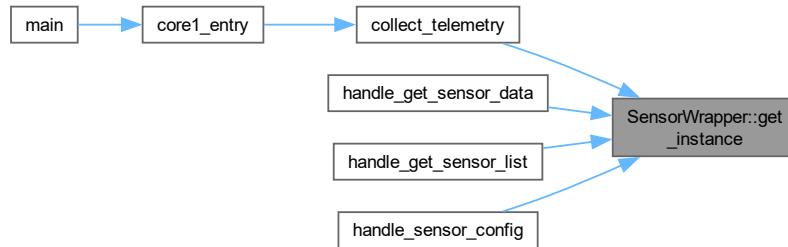
A reference to the single [SensorWrapper](#) instance.

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

Here is the call graph for this function:



Here is the caller graph for this function:



7.24.3.2 sensor_init()

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

Initializes a given sensor type on the specified I2C bus.

Parameters

<i>type</i>	The sensor type (LIGHT, ENVIRONMENT, etc.).
<i>i2c</i>	The I2C interface pointer.

Returns

True if initialization succeeded, otherwise false.

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

7.24.3.3 sensor_configure()

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

Configures an already initialized sensor with supplied settings.

Parameters

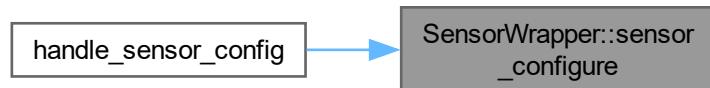
<i>type</i>	The sensor type.
<i>config</i>	Key-value pairs for sensor configuration.

Returns

True if the sensor was successfully configured, otherwise false.

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

Here is the caller graph for this function:

**7.24.3.4 sensor_read_data()**

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

Reads a specific data type (e.g., temperature, humidity) from a sensor.

Parameters

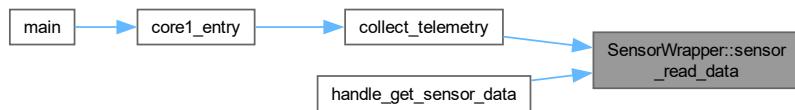
<code>sensorType</code>	The sensor type.
<code>dataType</code>	The type of data to read (light level, temperature, etc.).

Returns

The requested measurement. Returns 0.0f if sensor not found or uninitialized.

Definition at line 83 of file [ISensor.cpp](#).

Here is the caller graph for this function:



7.24.3.5 get_sensor()

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

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

7.24.3.6 scan_connected_sensors()

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

Scans the I2C bus for connected sensors.

Parameters

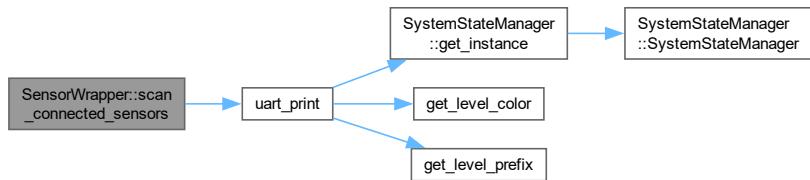
<i>i2c</i>	The I2C interface to scan
------------	---------------------------

Returns

A vector of pairs containing sensor type and I2C address of detected sensors

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

Here is the call graph for this function:



7.24.3.7 get_available_sensors()

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

Retrieves a list of available sensor types with their addresses.

Returns

A vector of pairs containing sensor type and I2C address

Definition at line 96 of file [ISensor.cpp](#).

Here is the caller graph for this function:



7.24.4 Member Data Documentation

7.24.4.1 sensors

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

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

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

- lib/sensors/[ISensor.h](#)
- lib/sensors/[ISensor.cpp](#)
- test/[test_mocks.cpp](#)

7.25 SystemStateManager Class Reference

Singleton class for managing global system states.

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

Collaboration diagram for SystemStateManager:



Public Member Functions

- bool [is_bootloader_reset_pending \(\) const](#)
Check if a bootloader reset is pending.
- void [set_bootloader_reset_pending \(bool pending\)](#)
Set the bootloader reset pending state.
- bool [is_gps_collection_paused \(\) const](#)
Check if GPS data collection is paused.
- void [set_gps_collection_paused \(bool paused\)](#)
Set the GPS collection paused state.
- bool [is_sd_card_mounted \(\) const](#)
Check if the SD card is currently mounted.
- void [set_sd_card_mounted \(bool mounted\)](#)
Set the SD card mounted state.
- [VerbosityLevel get_uart_verbosity \(\) const](#)
Get the current UART verbosity level.
- void [set_uart_verbosity \(VerbosityLevel level\)](#)
Set the UART verbosity level.
- [SystemStateManager \(const SystemStateManager &\)=delete](#)
- [SystemStateManager & operator= \(const SystemStateManager &\)=delete](#)

Static Public Member Functions

- static [SystemStateManager & get_instance \(\)](#)
Get the singleton instance of [SystemStateManager](#).

Private Member Functions

- [SystemStateManager \(\)](#)
Private constructor for singleton pattern.

Private Attributes

- bool [pending_bootloader_reset](#)
- bool [gps_collection_paused](#)
- bool [sd_card_mounted](#)
- [VerbosityLevel uart_verbosity](#)

Static Private Attributes

- static [SystemStateManager * instance = nullptr](#)
Pointer to the singleton instance.
- static mutex_t [instance_mutex](#)
Mutex for thread-safe access to the singleton instance.

7.25.1 Detailed Description

Singleton class for managing global system states.

This class provides a centralized, thread-safe way to manage system-wide state variables instead of using global variables. It ensures proper initialization and thread-safe access to system states across both cores.

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

7.25.2 Constructor & Destructor Documentation

7.25.2.1 SystemStateManager() [1/2]

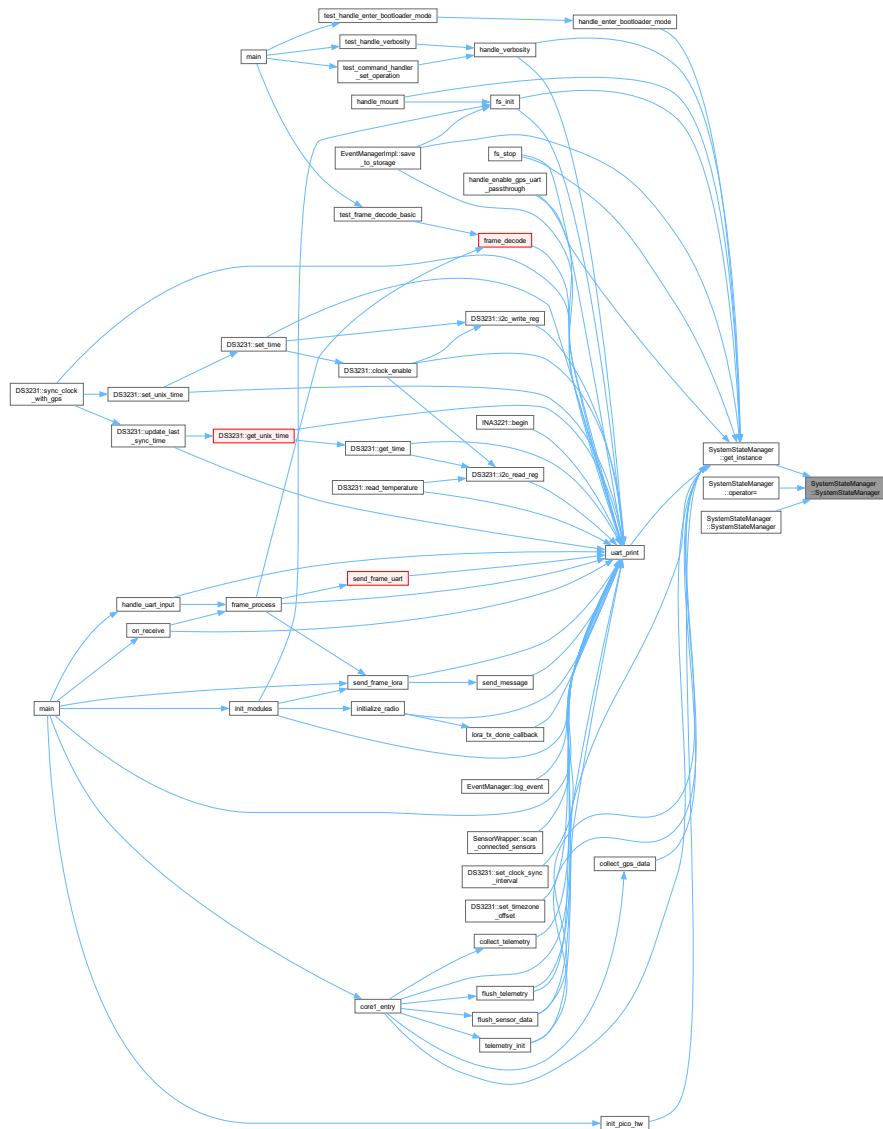
```
SystemStateManager::SystemStateManager () [inline], [private]
```

Private constructor for singleton pattern.

Initializes all state variables to their default values

Definition at line 43 of file [system_state_manager.h](#).

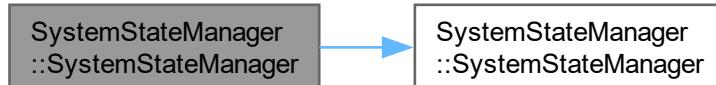
Here is the caller graph for this function:



7.25.2.2 SystemStateManager() [2/2]

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

Here is the call graph for this function:



7.25.3 Member Function Documentation

7.25.3.1 get_instance()

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

Get the singleton instance of [SystemStateManager](#).

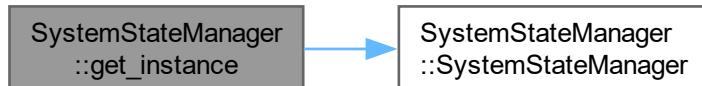
Returns

Reference to the singleton instance

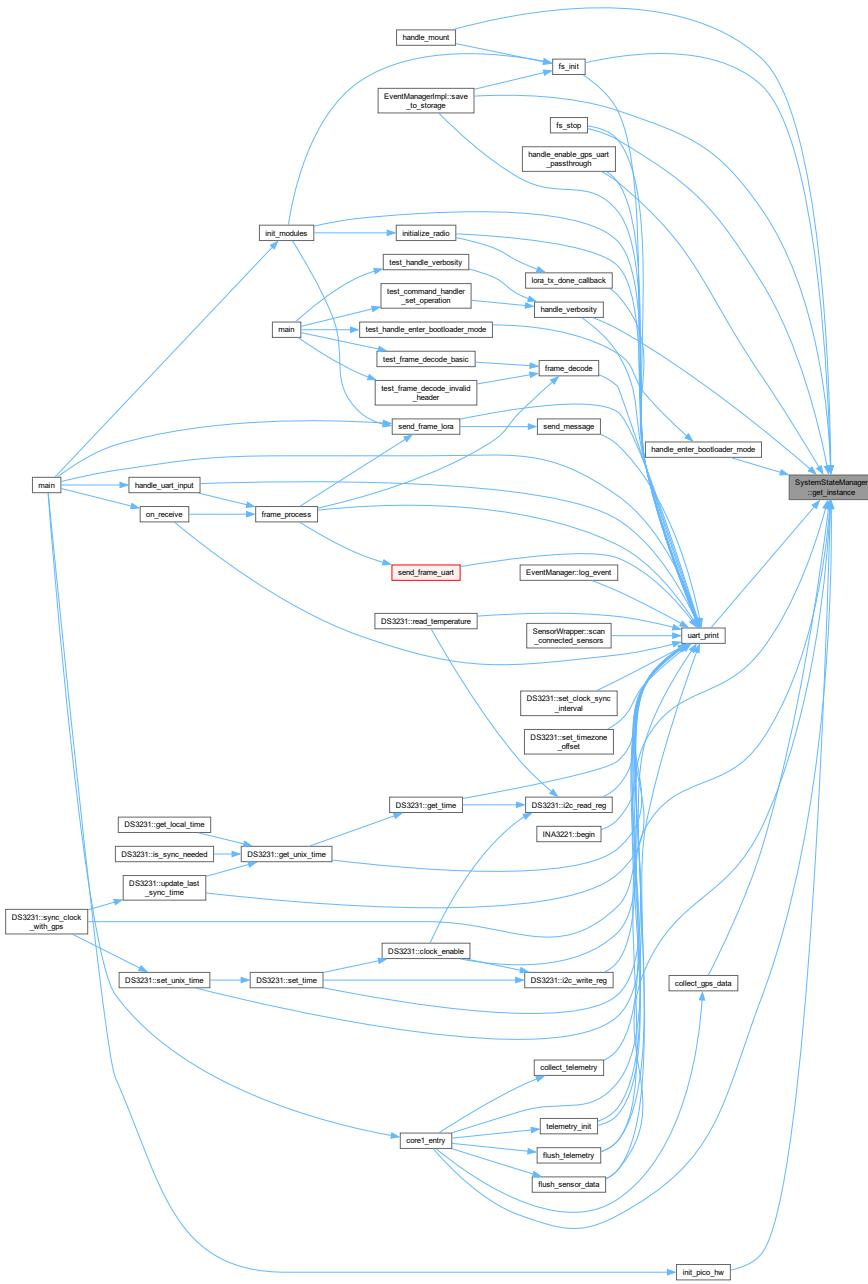
This method ensures thread-safe access to the singleton instance using a mutex. It initializes the mutex on first call and creates the instance if it doesn't exist.

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

Here is the call graph for this function:



Here is the caller graph for this function:



7.25.3.2 is_bootloader_reset_pending()

```
bool SystemStateManager::is_bootloader_reset_pending () const
```

Check if a bootloader reset is pending.

Returns

True if bootloader reset is pending, false otherwise

Definition at line 30 of file [system_state_manager.cpp](#).

7.25.3.3 set_bootloader_reset_pending()

```
void SystemStateManager::set_bootloader_reset_pending (
    bool pending)
```

Set the bootloader reset pending state.

Parameters

<i>pending</i>	The new state to set (true = reset pending)
----------------	---

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

Here is the caller graph for this function:



7.25.3.4 is_gps_collection_paused()

```
bool SystemStateManager::is_gps_collection_paused () const
```

Check if GPS data collection is paused.

Returns

True if GPS collection is paused, false otherwise

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

7.25.3.5 set_gps_collection_paused()

```
void SystemStateManager::set_gps_collection_paused (
    bool paused)
```

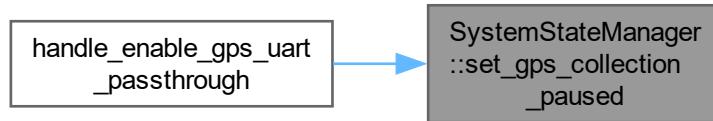
Set the GPS collection paused state.

Parameters

<i>paused</i>	The new state to set (true = collection paused)
---------------	---

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

Here is the caller graph for this function:



7.25.3.6 `is_sd_card_mounted()`

```
bool SystemStateManager::is_sd_card_mounted () const
```

Check if the SD card is currently mounted.

Returns

True if the SD card is mounted, false otherwise

Definition at line 46 of file [system_state_manager.cpp](#).

Here is the caller graph for this function:



7.25.3.7 `set_sd_card_mounted()`

```
void SystemStateManager::set_sd_card_mounted (
    bool mounted)
```

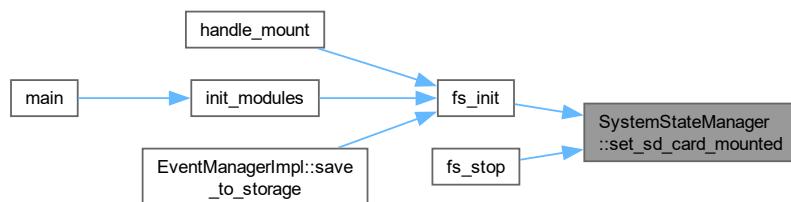
Set the SD card mounted state.

Parameters

<i>mounted</i>	The new state to set (true = SD card mounted)
----------------	---

Definition at line 50 of file [system_state_manager.cpp](#).

Here is the caller graph for this function:

**7.25.3.8 get_uart_verbosity()**

`VerbosityLevel SystemStateManager::get_uart_verbosity () const`

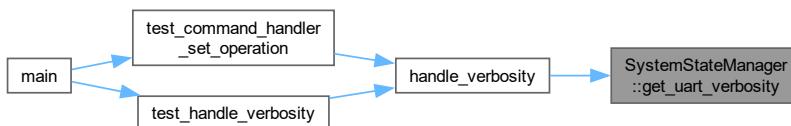
Get the current UART verbosity level.

Returns

The current verbosity level

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

Here is the caller graph for this function:

**7.25.3.9 set_uart_verbosity()**

```
void SystemStateManager::set_uart_verbosity (
    VerbosityLevel level)
```

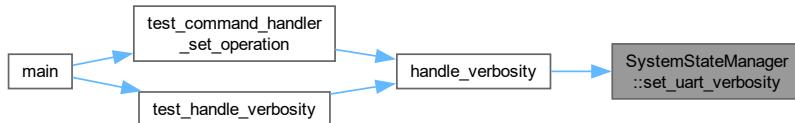
Set the UART verbosity level.

Parameters

<i>level</i>	The new verbosity level to set
--------------	--------------------------------

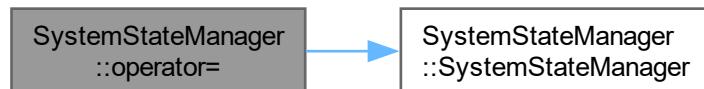
Definition at line 58 of file [system_state_manager.cpp](#).

Here is the caller graph for this function:

**7.25.3.10 operator=(*)**

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

Here is the call graph for this function:

**7.25.4 Member Data Documentation****7.25.4.1 instance**

```
SystemStateManager * SystemStateManager::instance = nullptr [static], [private]
```

Pointer to the singleton instance.

Definition at line 28 of file [system_state_manager.h](#).

7.25.4.2 instance_mutex

```
mutex_t SystemStateManager::instance_mutex [static], [private]
```

Mutex for thread-safe access to the singleton instance.

Definition at line 31 of file [system_state_manager.h](#).

7.25.4.3 pending_bootloader_reset

```
bool SystemStateManager::pending_bootloader_reset [private]
```

Flag indicating if a bootloader reset is pending

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

7.25.4.4 gps_collection_paused

```
bool SystemStateManager::gps_collection_paused [private]
```

Flag indicating if GPS data collection is paused

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

7.25.4.5 sd_card_mounted

```
bool SystemStateManager::sd_card_mounted [private]
```

Flag indicating if the SD card is currently mounted

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

7.25.4.6 uart_verbosity

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

Current verbosity level for UART logging

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

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

- lib/[system_state_manager.h](#)
- lib/[system_state_manager.cpp](#)

7.26 TelemetryRecord Struct Reference

Structure representing a single telemetry data point.

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

Public Member Functions

- std::string [to_csv \(\) const](#)

Public Attributes

- uint32_t `timestamp`
- std::string `build_version`
- float `battery_voltage`
- float `system_voltage`
- float `charge_current_usb`
- float `charge_current_solar`
- float `discharge_current`
- std::string `time`
- std::string `latitude`
- std::string `lat_dir`
- std::string `longitude`
- std::string `lon_dir`
- std::string `speed`
- std::string `course`
- std::string `date`
- std::string `fix_quality`
- std::string `satellites`
- std::string `altitude`

7.26.1 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 42 of file [telemetry_manager.h](#).

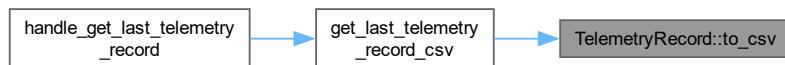
7.26.2 Member Function Documentation

7.26.2.1 `to_csv()`

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

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

Here is the caller graph for this function:



7.26.3 Member Data Documentation

7.26.3.1 timestamp

```
uint32_t TelemetryRecord::timestamp
```

Unix timestamp of the record

Definition at line 43 of file [telemetry_manager.h](#).

7.26.3.2 build_version

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

Build version of the firmware

Definition at line 45 of file [telemetry_manager.h](#).

7.26.3.3 battery_voltage

```
float TelemetryRecord::battery_voltage
```

Battery voltage in volts

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

7.26.3.4 system_voltage

```
float TelemetryRecord::system_voltage
```

System 5V rail voltage in volts

Definition at line 49 of file [telemetry_manager.h](#).

7.26.3.5 charge_current_usb

```
float TelemetryRecord::charge_current_usb
```

USB charging current in mA

Definition at line 50 of file [telemetry_manager.h](#).

7.26.3.6 charge_current_solar

```
float TelemetryRecord::charge_current_solar
```

Solar charging current in mA

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

7.26.3.7 `discharge_current`

```
float TelemetryRecord::discharge_current
```

Battery discharge current in mA

Definition at line 52 of file [telemetry_manager.h](#).

7.26.3.8 `time`

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

UTC time from GPS

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

7.26.3.9 `latitude`

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

Latitude from GPS

Definition at line 56 of file [telemetry_manager.h](#).

7.26.3.10 `lat_dir`

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

N/S latitude direction

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

7.26.3.11 `longitude`

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

Longitude from GPS

Definition at line 58 of file [telemetry_manager.h](#).

7.26.3.12 `lon_dir`

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

E/W longitude direction

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

7.26.3.13 speed

std::string TelemetryRecord::speed

Speed in knots

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

7.26.3.14 course

std::string TelemetryRecord::course

Course in degrees

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

7.26.3.15 date

std::string TelemetryRecord::date

Date from GPS

Definition at line 62 of file [telemetry_manager.h](#).

7.26.3.16 fix_quality

std::string TelemetryRecord::fix_quality

GPS fix quality

Definition at line 65 of file [telemetry_manager.h](#).

7.26.3.17 satellites

std::string TelemetryRecord::satellites

Number of satellites in view

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

7.26.3.18 altitude

std::string TelemetryRecord::altitude

Altitude in meters

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

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

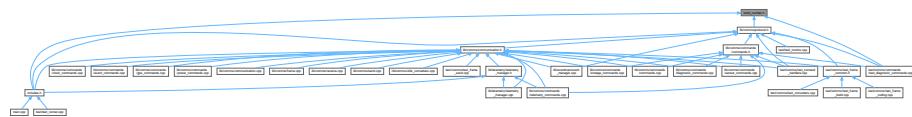
- lib/telemetry/[telemetry_manager.h](#)

Chapter 8

File Documentation

8.1 build_number.h File Reference

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



Macros

- `#define BUILD_NUMBER 456`

8.1.1 Macro Definition Documentation

8.1.1.1 BUILD_NUMBER

```
#define BUILD_NUMBER 456
```

Definition at line [6](#) of file [build_number.h](#).

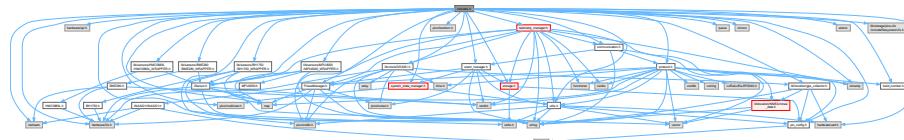
8.2 build_number.h

[Go to the documentation of this file.](#)

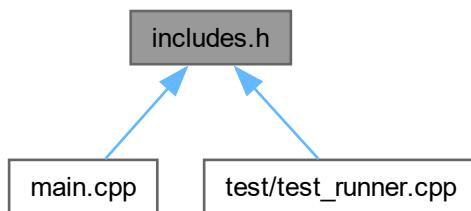
```
00001 //This file is automatically generated by build_number.cmake
00002
00003 #ifndef CMAKE_BUILD_NUMBER_HEADER
00004 #define CMAKE_BUILD_NUMBER_HEADER
00005
00006 #define BUILD_NUMBER 456
00007
00008 #endif
```

8.3 includes.h File Reference

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "hardware/i2c.h"
#include "hardware/uart.h"
#include "pico/multicore.h"
#include "event_manager.h"
#include "lib/powerman/PowerManager.h"
#include <pico/bootrom.h>
#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/sensors/HMC5883L/HMC5883L_WRAPPER.h"
#include "lib/sensors/MPU6050/MPU6050_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:



8.4 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/stl.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" // Corrected path
00012 #include <pico/bootrom.h>
00013
00014 #include "ISensor.h"
00015 #include "lib/sensors/BH1750/BH1750_WRAPPER.h" // Corrected path
00016 #include "lib/sensors/BME280/BME280_WRAPPER.h" // Corrected path
00017 #include "lib/sensors/HMC5883L/HMC5883L_WRAPPER.h" // Corrected path
00018 #include "lib/sensors/MPU6050/MPU6050_WRAPPER.h" // Corrected path
00019 #include "lib/clock/DS3231.h" // Corrected path
00020 #include <iostream>
00021 #include <iomanip>
00022 #include <queue>
00023 #include <chrono>
00024 #include "protocol.h"
00025 #include <atomic>
00026 #include <iostream>
00027 #include <map>
00028 #include "pin_config.h"
00029 #include "utils.h"
00030 #include "communication.h"
00031 #include "build_number.h"
00032 #include "lib/location/gps_collector.h"
00033 #include "lib/storage/storage.h" // Corrected path
00034 #include "lib/storage/pico-vfs/include/filesystem/vfs.h" // Corrected path
00035 #include "telemetry_manager.h"
00036 #include "system_state_manager.h"
00037
00038 #endif

```

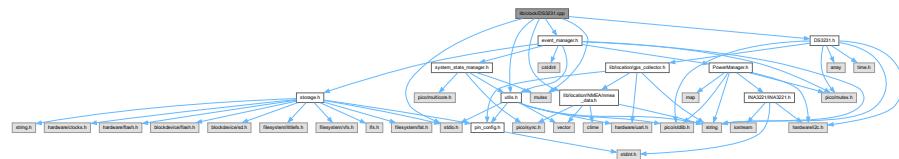
8.5 lib/clock/DS3231.cpp File Reference

```

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

```

Include dependency graph for DS3231.cpp:



8.6 DS3231.cpp

[Go to the documentation of this file.](#)

```

00001 #include "DS3231.h"
00002 #include "utils.h"

```

```

00003 #include <stdio.h>
00004 #include <mutex>
00005 #include "event_manager.h"
00006 #include "NMEA_data.h"
00007
0016 DS3231::DS3231(i2c_inst_t *i2c_instance) : i2c(i2c_instance), ds3231_addr(DS3231_DEVICE_ADDRESS) {
0017     recursive_mutex_init(&clock_mutex);
0018 }
0019
0020
0033 int DS3231::set_time(ds3231_data_t *data) {
0034     uint8_t temp[7] = {0};
0035
0036     if (clock_enable() != 0) {
0037         uart_print("Failed to enable clock oscillator", VerbosityLevel::ERROR);
0038         return -1;
0039     }
0040
0041     if (data->seconds > 59)
0042         data->seconds = 59;
0043     if (data->minutes > 59)
0044         data->minutes = 59;
0045     if (data->hours > 23)
0046         data->hours = 23;
0047     if (data->day > 7)
0048         data->day = 7;
0049     else if (data->day < 1)
0050         data->day = 1;
0051     if (data->date > 31)
0052         data->date = 31;
0053     else if (data->date < 1)
0054         data->date = 1;
0055     if (data->month > 12)
0056         data->month = 12;
0057     else if (data->month < 1)
0058         data->month = 1;
0059     if (data->year > 99)
0060         data->year = 99;
0061
0062     temp[0] = bin_to_bcd(data->seconds);
0063     temp[1] = bin_to_bcd(data->minutes);
0064     temp[2] = bin_to_bcd(data->hours);
0065     temp[2] &= ~(0x01 << 6); // Clear 12/24 hour bit
0066     temp[3] = bin_to_bcd(data->day);
0067     temp[4] = bin_to_bcd(data->date);
0068     temp[5] = bin_to_bcd(data->month);
0069     if (data->century)
0070         temp[5] |= (0x01 << 7);
0071     temp[6] = bin_to_bcd(data->year);
0072
0073     std::string status = "BCD values to be written to DS3231: " + std::to_string(temp[0]) + " " +
0074             std::to_string(temp[1]) + " " + std::to_string(temp[2]) + " " +
0075             std::to_string(temp[3]) + " " + std::to_string(temp[4]) + " " +
0076             std::to_string(temp[5]) + " " + std::to_string(temp[6]);
0077
0078     uart_print(status, VerbosityLevel::DEBUG);
0079
0080     int result = i2c_write_reg(DS3231_SECONDS_REG, 7, temp);
0081     if (result != 0) {
0082         uart_print("i2c write failed", VerbosityLevel::ERROR);
0083         return -1;
0084     }
0085
0086     return 0;
0087 }
0088
0089
00102 int DS3231::get_time(ds3231_data_t *data) {
00103     std::string status;
00104     uint8_t raw_data[7];
00105     int result = i2c_read_reg(DS3231_SECONDS_REG, 7, raw_data);
00106     if (result != 0) {
00107         status = "Failed to read time from DS3231";
00108         uart_print(status, VerbosityLevel::ERROR);
00109         return -1;
00110     }
00111
00112     data->seconds = bcd_to_bin(raw_data[0] & 0x7F); // Masking for CH bit (clock halt)
00113     data->minutes = bcd_to_bin(raw_data[1] & 0x7F);
00114     data->hours = bcd_to_bin(raw_data[2] & 0x3F); // Masking for 12/24 hour mode bit
00115     data->day = raw_data[3] & 0x07; // Day of week (1-7)
00116     data->date = bcd_to_bin(raw_data[4] & 0x3F);
00117     data->month = bcd_to_bin(raw_data[5] & 0x1F); // Masking for century bit
00118     data->century = (raw_data[5] & 0x80) >> 7;
00119     data->year = bcd_to_bin(raw_data[6]);
00120
00121     if (data->seconds > 59 || data->minutes > 59 || data->hours > 23 ||

```

```

00122     data->day < 1 || data->day > 7 || data->date < 1 || data->date > 31 ||
00123     data->month < 1 || data->month > 12 || data->year > 99) {
00124     uart_print("Invalid data read from DS3231", VerbosityLevel::ERROR);
00125     return -1;
00126 }
00127
00128 return 0;
00129 }
00130
00131
00143 int DS3231::read_temperature(float *resolution) {
00144     std::string status;
00145     uint8_t temp[2];
00146     int result = i2c_read_reg(DS3231_TEMPERATURE_MSB_REG, 2, temp);
00147     if (result != 0) {
00148         status = "Failed to read temperature from DS3231";
00149         uart_print(status, VerbosityLevel::ERROR);
00150         return -1;
00151     }
00152
00153     int8_t temperature_msb = (int8_t)temp[0];
00154     uint8_t temperature_lsb = temp[1] >> 6; // Only the 2 MSB are valid
00155
00156     *resolution = temperature_msb + (temperature_lsb * 0.25f); // 0.25 degree resolution
00157
00158     return 0;
00159 }
00160
00161
00172 int DS3231::set_unix_time(time_t unix_time) {
00173     struct tm *timeinfo = gmtime(&unix_time);
00174     if (timeinfo == NULL) {
00175         uart_print("Error: gmtime() failed", VerbosityLevel::ERROR);
00176         return -1;
00177     }
00178
00179     ds3231_data_t data;
00180     data.seconds = timeinfo->tm_sec;
00181     data.minutes = timeinfo->tm_min;
00182     data.hours = timeinfo->tm_hour;
00183     data.day = timeinfo->tm_wday == 0 ? 7 : timeinfo->tm_wday; // Sunday is 0 in tm struct, but 1 in
00184     DS3231
00185     data.date = timeinfo->tm_mday;
00186     data.month = timeinfo->tm_mon + 1; // Month is 0-11 in tm struct, but 1-12 in DS3231
00187     data.year = timeinfo->tm_year - 100; // Year is since 1900, we want the last two digits
00188     data.century = timeinfo->tm_year >= 2000;
00189
00190     return set_time(&data);
00191 }
00192
00202 time_t DS3231::get_unix_time() {
00203     ds3231_data_t data;
00204     if (get_time(&data)) {
00205         return -1;
00206     }
00207
00208     struct tm timeinfo;
00209     timeinfo.tm_sec = data.seconds;
00210     timeinfo.tm_min = data.minutes;
00211     timeinfo.tm_hour = data.hours;
00212     timeinfo.tm_mday = data.date;
00213     timeinfo.tm_mon = data.month - 1; // Month is 0-11 in tm struct, but 1-12 in DS3231
00214     timeinfo.tm_year = data.year + 100; // Year is since 1900
00215
00216     // mktime assumes that tm_wday and tm_yday are uninitialized
00217     timeinfo.tm_wday = 0;
00218     timeinfo.tm_yday = 0;
00219     timeinfo.tm_isdst = 0; // Set to 0 to use UTC
00220
00221     time_t timestamp = mktime(&timeinfo);
00222     if (timestamp == (time_t)(-1)) {
00223         uart_print("Error: mktime() failed", VerbosityLevel::ERROR);
00224         return -1;
00225     }
00226
00227     return timestamp;
00228 }
00229
00230
00239 int DS3231::clock_enable() {
00240     std::string status;
00241     uint8_t control_reg = 0;
00242     int result = i2c_read_reg(DS3231_CONTROL_REG, 1, &control_reg);
00243     if (result != 0) {
00244         status = "Failed to read control register";
00245         uart_print(status, VerbosityLevel::ERROR);

```

```

00246     return -1;
00247 }
00248
00249 // Clear the EOSC bit to enable the oscillator
00250 control_reg &= ~(1 << 7);
00251
00252 result = i2c_write_reg(DS3231_CONTROL_REG, 1, &control_reg);
00253 if (result != 0) {
00254     status = "Failed to write control register";
00255     uart_print(status, VerbosityLevel::ERROR);
00256     return -1;
00257 }
00258
00259 return 0;
00260 }
00261
00262
00271 int16_t DS3231::get_timezone_offset() const {
00272     return timezone_offset_minutes_;
00273 }
00274
00275
00286 void DS3231::set_timezone_offset(int16_t offset_minutes) {
00287     // Validate range: -12 hours to +12 hours (-720 to +720 minutes)
00288     if (offset_minutes >= -720 && offset_minutes <= 720) {
00289         timezone_offset_minutes_ = offset_minutes;
00290     } else {
00291         uart_print("Error: Invalid timezone offset", VerbosityLevel::ERROR);
00292     }
00293 }
00294
00295
00303 uint32_t DS3231::get_clock_sync_interval() const {
00304     return sync_interval_minutes_;
00305 }
00306
00307
00317 void DS3231::set_clock_sync_interval(uint32_t interval_minutes) {
00318     if (interval_minutes >= 1 && interval_minutes <= 43200) {
00319         sync_interval_minutes_ = interval_minutes;
00320     } else {
00321         uart_print("Error: Invalid sync interval", VerbosityLevel::ERROR);
00322     }
00323 }
00324
00325
00334 time_t DS3231::get_last_sync_time() const {
00335     return last_sync_time_;
00336 }
00337
00338
00347 void DS3231::update_last_sync_time() {
00348     last_sync_time_ = get_unix_time();
00349     uart_print("Clock sync time updated: " + std::to_string(last_sync_time_), VerbosityLevel::INFO);
00350 }
00351
00352
00360 time_t DS3231::get_local_time() {
00361     time_t utc_time = get_unix_time();
00362     if (utc_time == -1) {
00363         return -1;
00364     }
00365
00366     return utc_time + (timezone_offset_minutes_ * 60);
00367 }
00368
00369
00379 bool DS3231::is_sync_needed() {
00380     if (last_sync_time_ == 0) {
00381         return true;
00382     }
00383
00384     time_t current_time = get_unix_time();
00385     if (current_time == -1) {
00386         return true;
00387     }
00388
00389     time_t time_since_last_sync = current_time - last_sync_time_;
00390     uint32_t minutes_since_last_sync = time_since_last_sync / 60;
00391
00392     return minutes_since_last_sync >= sync_interval_minutes_;
00393 }
00394
00395
00409 bool DS3231::sync_clock_with_gps() {
00410     extern NMEAData nmea_data;
00411

```

```

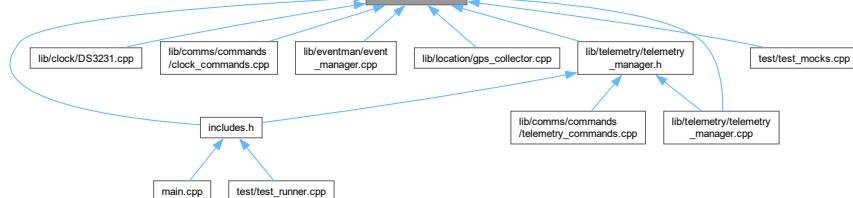
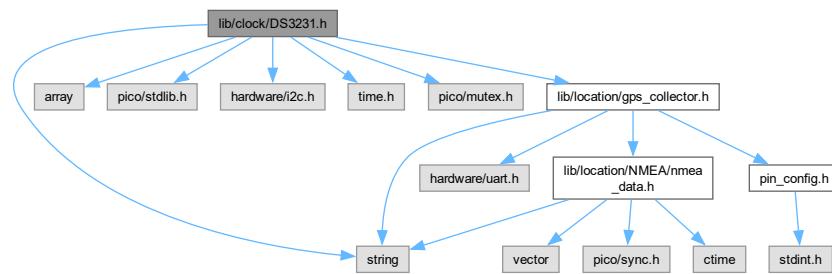
00412     if (!nmea_data.has_valid_time()) {
00413         uart_print("GPS time data not available for sync", VerbosityLevel::WARNING);
00414         EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
00415         return false;
00416     }
00417
00418     time_t gps_time = nmea_data.get_unix_time();
00419     if (gps_time <= 0) {
00420         uart_print("Invalid GPS time for sync", VerbosityLevel::ERROR);
00421         EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
00422         return false;
00423     }
00424
00425     if (set_unix_time(gps_time) != 0) {
00426         uart_print("Failed to set system time from GPS", VerbosityLevel::ERROR);
00427         EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
00428         return false;
00429     }
00430
00431     update_last_sync_time();
00432
00433     EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC);
00434     uart_print("Clock synced with GPS time: " + std::to_string(gps_time), VerbosityLevel::INFO);
00435
00436     return true;
00437 }
00438
00439 // ===== private methods
00440
00445 int DS3231::i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
00446     if (!length)
00447         return -1;
00448
00449     std::string status = "Reading register " + std::to_string(reg_addr) + " from DS3231";
00450     uart_print(status, VerbosityLevel::DEBUG);
00451     recursive_mutex_enter_blocking(&clock_mutex_);
00452     uint8_t reg = reg_addr;
00453     int write_result = i2c_write_blocking(i2c, ds3231_addr, &reg, 1, true);
00454     if (write_result == PICO_ERROR_GENERIC) {
00455         status = "Failed to write register address to DS3231";
00456         uart_print(status, VerbosityLevel::ERROR);
00457         recursive_mutex_exit(&clock_mutex_);
00458         return -1;
00459     }
00460     int read_result = i2c_read_blocking(i2c, ds3231_addr, data, length, false);
00461     if (read_result == PICO_ERROR_GENERIC) {
00462         status = "Failed to read register data from DS3231";
00463         uart_print(status, VerbosityLevel::ERROR);
00464         recursive_mutex_exit(&clock_mutex_);
00465         return -1;
00466     }
00467     recursive_mutex_exit(&clock_mutex_);
00468
00469     return 0;
00470 }
00471
00496 int DS3231::i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
00497     if (!length)
00498         return -1;
00499
00500     recursive_mutex_enter_blocking(&clock_mutex_);
00501     std::vector<uint8_t> message(length + 1);
00502     message[0] = reg_addr;
00503     for (int i = 0; i < length; i++) {
00504         message[i + 1] = data[i];
00505     }
00506     int write_result = i2c_write_blocking(i2c, ds3231_addr, message.data(), (length + 1), false);
00507     if (write_result == PICO_ERROR_GENERIC) {
00508         uart_print("Error: i2c_write_blocking failed in i2c_write_reg", VerbosityLevel::ERROR);
00509         recursive_mutex_exit(&clock_mutex_);
00510         return -1;
00511     }
00512     recursive_mutex_exit(&clock_mutex_);
00513
00514     return 0;
00515 }
00516
00526 uint8_t DS3231::bin_to_bcd(const uint8_t data) {
00527     uint8_t ones_digit = (uint8_t)(data % 10);
00528     uint8_t tens_digit = (uint8_t)(data - ones_digit) / 10;
00529     return ((tens_digit << 4) + ones_digit);
00530 }
00531
00532
00542 uint8_t DS3231::bcd_to_bin(const uint8_t bcd) {
00543     uint8_t ones_digit = (uint8_t)(bcd & 0x0F);
00544     uint8_t tens_digit = (uint8_t)(bcd >> 4);

```

```
00545     return (tens_digit * 10 + ones_digit);
00546 } // End of DS3231_RTC group
```

8.7 lib/clock/DS3231.h File Reference

```
#include <string>
#include <array>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include <time.h>
#include "pico/mutex.h"
#include "lib/location/gps_collector.h"
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.

8.7.1 Macro Definition Documentation

8.7.1.1 DS3231_DEVICE_ADDRESS

```
#define DS3231_DEVICE_ADDRESS 0x68
```

`DS3231` I2C device address.

Definition at line 15 of file [DS3231.h](#).

8.7.1.2 DS3231_SECONDS_REG

```
#define DS3231_SECONDS_REG 0x00
```

Register address: Seconds (0-59)

Definition at line 20 of file [DS3231.h](#).

8.7.1.3 DS3231_MINUTES_REG

```
#define DS3231_MINUTES_REG 0x01
```

Register address: Minutes (0-59)

Definition at line [25](#) of file [DS3231.h](#).

8.7.1.4 DS3231_HOURS_REG

```
#define DS3231_HOURS_REG 0x02
```

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

Definition at line [30](#) of file [DS3231.h](#).

8.7.1.5 DS3231_DAY_REG

```
#define DS3231_DAY_REG 0x03
```

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

Definition at line [35](#) of file [DS3231.h](#).

8.7.1.6 DS3231_DATE_REG

```
#define DS3231_DATE_REG 0x04
```

Register address: Date (1-31)

Definition at line [40](#) of file [DS3231.h](#).

8.7.1.7 DS3231_MONTH_REG

```
#define DS3231_MONTH_REG 0x05
```

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

Definition at line [45](#) of file [DS3231.h](#).

8.7.1.8 DS3231_YEAR_REG

```
#define DS3231_YEAR_REG 0x06
```

Register address: Year (00-99)

Definition at line [50](#) of file [DS3231.h](#).

8.7.1.9 DS3231_CONTROL_REG

```
#define DS3231_CONTROL_REG 0x0E
```

Register address: Control register.

Definition at line 55 of file [DS3231.h](#).

8.7.1.10 DS3231_CONTROL_STATUS_REG

```
#define DS3231_CONTROL_STATUS_REG 0x0F
```

Register address: Control/Status register.

Definition at line 60 of file [DS3231.h](#).

8.7.1.11 DS3231_TEMPERATURE_MSB_REG

```
#define DS3231_TEMPERATURE_MSB_REG 0x11
```

Register address: Temperature register (MSB)

Definition at line 65 of file [DS3231.h](#).

8.7.1.12 DS3231_TEMPERATURE_LSB_REG

```
#define DS3231_TEMPERATURE_LSB_REG 0x12
```

Register address: Temperature register (LSB)

Definition at line 70 of file [DS3231.h](#).

8.7.2 Enumeration Type Documentation

8.7.2.1 days_of_week

```
enum days_of_week
```

Enumeration of days of the week.

Enumerator

MONDAY	Monday.
TUESDAY	Tuesday.
WEDNESDAY	Wednesday.
THURSDAY	Thursday.
FRIDAY	Friday.
SATURDAY	Saturday.
SUNDAY	Sunday.

Definition at line 76 of file [DS3231.h](#).

8.8 DS3231.h

[Go to the documentation of this file.](#)

```

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);
00116
00123     int set_time(ds3231_data_t *data);
00124
00131     int get_time(ds3231_data_t *data);
00132
00139     int read_temperature(float *resolution);
00140
00147     int set_unix_time(time_t unix_time);
00148
00154     time_t get_unix_time();
00155
00161     int clock_enable();
00162
00168     int16_t get_timezone_offset() const;
00169
00175     void set_timezone_offset(int16_t offset_minutes);
00176
00182     uint32_t get_clock_sync_interval() const;
00183
00189     void set_clock_sync_interval(uint32_t interval_minutes);
00190
00196     time_t get_last_sync_time() const;
00197

```

```

00201     void update_last_sync_time();
00202
00208     time_t get_local_time();
00209
00215     bool is_sync_needed();
00216
00222     bool sync_clock_with_gps();
00223
00224
00225 private:
00226     i2c_inst_t *i2c;
00227     uint8_t ds3231_addr;
00228
00237     int i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data);
00238
00247     int i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data);
00248
00255     uint8_t bin_to_bcd(const uint8_t data);
00256
00263     uint8_t bcd_to_bin(const uint8_t bcd);
00264
00265     recursive_mutex_t clock_mutex_;
00266     int16_t timezone_offset_minutes_ = 60;
00267     uint32_t sync_interval_minutes_ = 1440;
00268     time_t last_sync_time_ = 0;
00269 }
00270
00271 #endif // DS3231_H

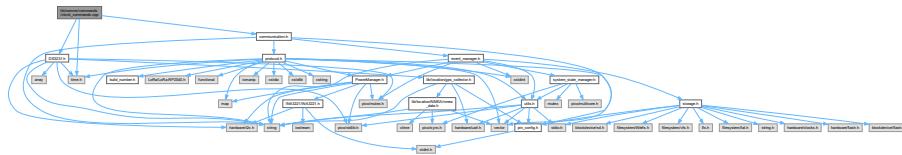
```

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

```

#include "communication.h"
#include <time.h>
#include "DS3231.h"
Include dependency graph for clock_commands.cpp:

```



Macros

- #define CLOCK_GROUP 3
- #define TIME 0
- #define TIMEZONE_OFFSET 1
- #define CLOCK_SYNC_INTERVAL 2
- #define LAST_SYNC_TIME 3

Functions

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

Handler for getting and setting clock synchronization interval.
- std::vector< Frame > handle_get_last_sync_time (const std::string ¶m, OperationType operationType)

Handler for getting last clock sync time.

Variables

- DS3231 systemClock

8.9.1 Macro Definition Documentation**8.9.1.1 CLOCK_GROUP**

```
#define CLOCK_GROUP 3
```

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

8.9.1.2 TIME

```
#define TIME 0
```

Definition at line 6 of file [clock_commands.cpp](#).

8.9.1.3 TIMEZONE_OFFSET

```
#define TIMEZONE_OFFSET 1
```

Definition at line 7 of file [clock_commands.cpp](#).

8.9.1.4 CLOCK_SYNC_INTERVAL

```
#define CLOCK_SYNC_INTERVAL 2
```

Definition at line 8 of file [clock_commands.cpp](#).

8.9.1.5 LAST_SYNC_TIME

```
#define LAST_SYNC_TIME 3
```

Definition at line 9 of file [clock_commands.cpp](#).

8.10 clock_commands.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002 #include <time.h>
00003 #include "DS3231.h" // Include the DS3231 header
00004
00005 #define CLOCK_GROUP 3
00006 #define TIME 0
00007 #define TIMEZONE_OFFSET 1
00008 #define CLOCK_SYNC_INTERVAL 2
00009 #define LAST_SYNC_TIME 3
0010
0016
0017 extern DS3231 systemClock;
0018
0032 std::vector<Frame> handle_time(const std::string& param, OperationType operationType) {
0033     std::vector<Frame> frames;
0034     std::string error_msg;
0035
0036     if (operationType == OperationType::SET) {
0037         if (param.empty()) {
0038             error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
0039             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0040             return frames;
0041         }
0042         try {
0043             time_t newTime = std::stoll(param);
0044             if (newTime <= 0) {
0045                 error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
0046                 frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0047                 return frames;
0048             }
0049
0050             if (systemClock.set_unix_time(newTime) != 0) {
0051                 error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
0052                 frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0053                 return frames;
0054             }
0055
0056             EventEmitter::emit(EventGroup::CLOCK, ClockEvent::CHANGED);
0057             frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, TIME,
0058                 std::to_string(systemClock.get_unix_time())));
0059             return frames;
0060         } catch (...) {
0061             error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
0062             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0063             return frames;
0064         }
0064     } else if (operationType == OperationType::GET) {
0065         if (!param.empty()) {
0066             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
0067             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0068             return frames;
0069         }
0070
0071         uint32_t time_unix = systemClock.get_local_time();
0072         if (time_unix == 0) {
0073             error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
0074             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0075             return frames;
0076         }
0077
0078         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, TIME,
0079             std::to_string(time_unix)));
0080         return frames;
0081
0082     error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
0083     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
0084     return frames;
0085 }
0086
0097 std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType) {
0098     std::vector<Frame> frames;
0099     std::string error_msg;
0100
0101     if (!(operationType == OperationType::GET || operationType == OperationType::SET)) {
0102         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
0103         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
0104         return frames;
0105     }
0106
0107     if (operationType == OperationType::GET) {
0108         if (!param.empty()) {

```

```

00109         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00110         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET,
00111             error_msg));
00112     }
00113
00114     int offset = systemClock.get_timezone_offset();
00115     std::string offset_set = std::to_string(offset);
00116     frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, TIMEZONE_OFFSET, offset_set));
00117     return frames;
00118 }
00119
00120 if (param.empty()) {
00121     error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00122     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
00123     return frames;
00124 }
00125
00126 try {
00127     int16_t offset = std::stoi(param);
00128     if (offset < -720 || offset > 720) {
00129         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00130         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET,
00131             error_msg));
00132         return frames;
00133     }
00134     systemClock.set_timezone_offset(offset);
00135     std::string offset_set = std::to_string(offset);
00136     frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, TIMEZONE_OFFSET, offset_set));
00137     return frames;
00138 } catch (...) {
00139     error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00140     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
00141     return frames;
00142 }
00143 }
00144
00145
00156 std::vector<Frame> handle_clock_sync_interval(const std::string& param, OperationType operationType) {
00157     std::vector<Frame> frames;
00158     std::string error_msg;
00159
00160     if (!(operationType == OperationType::GET || operationType == OperationType::SET)) {
00161         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00162         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
00163             error_msg));
00164         return frames;
00165     }
00166
00167     if (operationType == OperationType::GET) {
00168         if (!param.empty()) {
00169             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00170             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
00171                 error_msg));
00172             return frames;
00173         }
00174         uint32_t syncInterval = systemClock.get_clock_sync_interval();
00175         std::string clockSyncInterval = std::to_string(syncInterval);
00176         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
00177             clockSyncInterval));
00178         return frames;
00179     }
00180     if (operationType == OperationType::SET) {
00181         if (param.empty()) {
00182             error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00183             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
00184                 error_msg));
00185             return frames;
00186         }
00187         try {
00188             uint32_t interval = std::stoul(param);
00189             systemClock.set_clock_sync_interval(interval);
00190             std::string interval_set = std::to_string(interval);
00191             frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
00192                 interval_set));
00193             return frames;
00194         } catch (...) {
00195             error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00196             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
00197                 error_msg));
00198         }
00199     }
00200 }
```

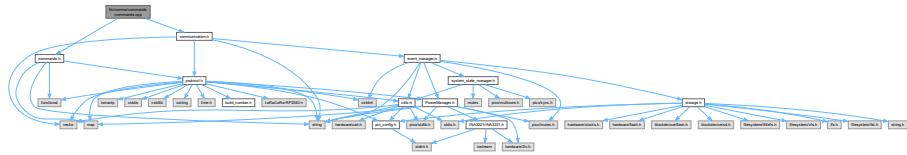
```

00198     error_msg = error_code_to_string(ErrorCode::UNKNOWN_ERROR);
00199     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
00200     return frames;
00201 }
00202
00212 std::vector<Frame> handle_get_last_sync_time(const std::string& param, OperationType operationType) {
00213     std::vector<Frame> frames;
00214     std::string error_msg;
00215
00216     if (operationType != OperationType::GET || !param.empty()) {
00217         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00218         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, LAST_SYNC_TIME, error_msg));
00219         return frames;
00220     }
00221
00222     time_t lastSyncTime = systemClock.get_last_sync_time();
00223
00224     if (lastSyncTime == 0) {
00225         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, LAST_SYNC_TIME, "NEVER"));
00226     } else {
00227         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, LAST_SYNC_TIME,
00228                                     std::to_string(lastSyncTime)));
00229     }
00230
00231     return frames;
00232 } // end of ClockCommands group

```

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

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



Typedefs

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

Functions

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

Variables

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

8.12 commands.cpp

[Go to the documentation of this file.](#)

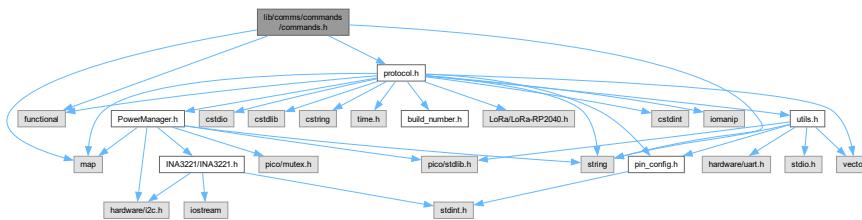
```

00001 // commands/commands.cpp
00002 #include "commands.h"
00003 #include "communication.h"
00004
00010
00015 using CommandHandler = std::function<std::vector<Frame>(&const std::string, OperationType)>;
00016
00021 using CommandMap = std::map<uint32_t, CommandHandler>;
00022
00027 CommandMap command_handlers = {
00028     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(0)), handle_get_commands_list},
00029     // Group 1, Command 0
00030     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(1)), handle_get_build_version},
00031     // Group 1, Command 1
00032     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(8)), handle_verbosity},
00033     // Group 1, Command 9
00034     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(9)), handle_enter_bootloader_mode},
00035     // Group 1, Command 9
00036     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(0)), handle_get_power_manager_ids},
00037     // Group 2, Command 0
00038     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(2)), handle_get_voltage_battery},
00039     // Group 2, Command 2
00040     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(3)), handle_get_voltage_5v},
00041     // Group 2, Command 3
00042     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(4)), handle_get_current_charge_usb},
00043     // Group 2, Command 4
00044     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(5)), handle_get_current_charge_solar},
00045     // Group 2, Command 5
00046     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(6)), handle_get_current_charge_total},
00047     // Group 2, Command 6
00048     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(7)), handle_get_current_draw},
00049     // Group 2, Command 7
00050     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(0)), handle_time},
00051     // Group 3, Command 0
00052     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(1)), handle_timezone_offset},
00053     // Group 3, Command 1
00054     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(2)), handle_clock_sync_interval},
00055     // Group 3, Command 2
00056     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(3)), handle_get_last_sync_time},
00057     // Group 3, Command 3
00058     {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(0)), handle_get_sensor_data},
00059     // Group 4, Command 0
00060     {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(1)), handle_sensor_config},
00061     // Group 4, Command 1
00062     {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(3)), handle_get_sensor_list},
00063     // Group 4, Command 3
00064     {((static_cast<uint32_t>(5) << 8) | static_cast<uint32_t>(1)), handle_get_last_events},
00065     // Group 5, Command 1
00066     {((static_cast<uint32_t>(5) << 8) | static_cast<uint32_t>(2)), handle_get_event_count},
00067     // Group 5, Command 2
00068     {((static_cast<uint32_t>(6) << 8) | static_cast<uint32_t>(0)), handle_list_files},
00069     // Group 6, Command 0
00070     {((static_cast<uint32_t>(6) << 8) | static_cast<uint32_t>(4)), handle_mount},
00071     // Group 6, Command 4
00072     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(1)), handle_gps_power_status},
00073     // Group 7, Command 1
00074     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(2)), handle_enable_gps_uart_passthrough},
00075     // Group 7, Command 3
00076     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(3)), handle_get_rmc_data},
00077     // Group 7, Command 4
00078     {((static_cast<uint32_t>(8) << 8) | static_cast<uint32_t>(2)), handle_get_last_telemetry_record},
00079     {((static_cast<uint32_t>(8) << 8) | static_cast<uint32_t>(3)), handle_get_last_sensor_record},
00080 };
00081
00082
00083 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
00084 operationType) {
00085     auto it = command_handlers.find(commandKey);
00086     if (it != command_handlers.end()) {
00087         CommandHandler handler = it->second;
00088         return handler(param, operationType);
00089     } else {
00090         std::vector<Frame> frames;
00091         frames.push_back(frame_build(OperationType::ERR, 0, 0, "INVALID COMMAND"));
00092         return frames;
00093     }
00094 } // end of CommandSystem group

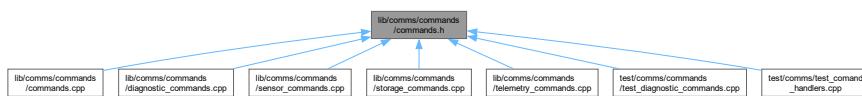
```

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

```
#include <string>
#include <functional>
#include <map>
#include "protocol.h"
Include dependency graph for commands.h:
```



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



Functions

- `std::vector< Frame > handle_time (const std::string ¶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_clock_sync_interval (const std::string ¶m, OperationType operationType)`
Handler for getting and setting clock synchronization interval.
- `std::vector< Frame > handle_get_last_sync_time (const std::string ¶m, OperationType operationType)`
Handler for getting last clock sync time.
- `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_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_rmc_data (const std::string ¶m, OperationType operationType)`

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

Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.
- std::vector< Frame > **handle_get_power_manager_ids** (const std::string ¶m, OperationType operationType)

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

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

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

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

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

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

Handler for getting system current draw.
- 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_list_files** (const std::string ¶m, OperationType operationType)

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

Handles the SD card mount/unmount command.
- std::vector< Frame > **handle_get_sensor_data** (const std::string ¶m, OperationType operationType)

Handler for reading sensor data.
- std::vector< Frame > **handle_sensor_config** (const std::string ¶m, OperationType operationType)

Handler for configuring sensors.
- std::vector< Frame > **handle_get_sensor_list** (const std::string ¶m, OperationType operationType)

Handler for listing available sensors.
- 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.

8.14 commands.h

[Go to the documentation of this file.](#)

```

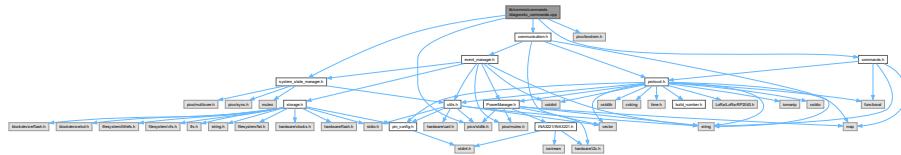
00001 // commands/commands.h
00002 #ifndef COMMANDS_H
00003 #define COMMANDS_H
00004
00005 #include <string>
00006 #include <functional>
00007 #include <map>
00008 #include "protocol.h"
00009
00010 // CLOCK
00011 std::vector<Frame> handle_time(const std::string& param, OperationType operationType);
00012 std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType);
00013 std::vector<Frame> handle_clock_sync_interval(const std::string& param, OperationType operationType);
00014 std::vector<Frame> handle_get_last_sync_time(const std::string& param, OperationType operationType);
00015
00016
00017 // DIAG
00018 std::vector<Frame> handle_get_commands_list(const std::string& param, OperationType operationType);
00019 std::vector<Frame> handle_get_build_version(const std::string& param, OperationType operationType);
00020 std::vector<Frame> handle_verbosity(const std::string& param, OperationType operationType);
00021 std::vector<Frame> handle_enter_bootloader_mode(const std::string& param, OperationType
00022 operationType);
00023
00024 // GPS
00025 std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType);
00026 std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType
00027 operationType);
00028 std::vector<Frame> handle_get_rmc_data(const std::string& param, OperationType operationType);
00029 std::vector<Frame> handle_get_gga_data(const std::string& param, OperationType operationType);
00030
00031 // POWER
00032 std::vector<Frame> handle_get_power_manager_ids(const std::string& param, OperationType
00033 operationType);
00034 std::vector<Frame> handle_get_voltage_battery(const std::string& param, OperationType operationType);
00035 std::vector<Frame> handle_get_voltage_5v(const std::string& param, OperationType operationType);
00036 std::vector<Frame> handle_get_current_charge_usb(const std::string& param, OperationType
00037 operationType);
00038 std::vector<Frame> handle_get_current_charge_solar(const std::string& param, OperationType
00039 operationType);
00040 std::vector<Frame> handle_get_current_charge_total(const std::string& param, OperationType
00041 operationType);
00042 std::vector<Frame> handle_get_current_draw(const std::string& param, OperationType operationType);
00043
00044
00045
00046 // STORAGE
00047 std::vector<Frame> handle_list_files(const std::string& param, OperationType operationType);
00048 std::vector<Frame> handle_mount(const std::string& param, OperationType operationType);
00049
00050 // SENSOR
00051 std::vector<Frame> handle_get_sensor_data(const std::string& param, OperationType operationType);
00052 std::vector<Frame> handle_sensor_config(const std::string& param, OperationType operationType);
00053 std::vector<Frame> handle_get_sensor_list(const std::string& param, OperationType operationType);
00054
00055
00056 // TELEMETRY
00057 std::vector<Frame> handle_get_last_telemetry_record(const std::string& param, OperationType
00058 operationType);
00059 std::vector<Frame> handle_get_last_sensor_record(const std::string& param, OperationType
00060 operationType);
00061 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
00062 operationType);
00063 extern std::map<uint32_t, std::function<std::vector<Frame>(const std::string&, OperationType)>>
00064 command_handlers;
00065
00066 #endif

```

8.15 lib/comms/commands/diagnostic_commands.cpp File Reference

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

```
#include "pico/stdlib.h"
#include "pico/bootrom.h"
#include "system_state_manager.h"
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_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.

8.16 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
00021 std::vector<Frame> handle_get_commands_list(const std::string& param, OperationType operationType) {
00022     std::vector<Frame> frames;
00023     std::string error_msg;
00024
00025     if (!param.empty()) {
00026         error_msg = error\_code\_to\_string(ErrorCode::PARAM_UNNECESSARY);
00027         frames.push_back(frame\_build(OperationType::ERR, 1, 0, error_msg));
00028         return frames;
00029     }
00030
00031     if (!(operationType == OperationType::GET)) {
00032         error_msg = error\_code\_to\_string(ErrorCode::INVALID_OPERATION);
00033         frames.push_back(frame\_build(OperationType::ERR, 1, 0, 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, 1, 0, combined_command_details));
00047             combined_command_details = "";
00048         }
00049
00050         if (!combined_command_details.empty()) {
00051             combined_command_details += "-";
00052         }
00053     }
00054 }
```

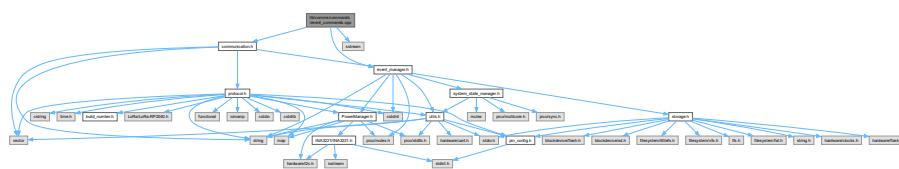
```

00053     combined_command_details += command_details;
00054 }
00055
00056 if (!combined_command_details.empty()) {
00057     frames.push_back(frame_build(OperationType::SEQ, 1, 0, combined_command_details));
00058 }
00059
00060 frames.push_back(frame_build(OperationType::VAL, 1, 0, "SEQ_DONE"));
00061 return frames;
00062 }
00063
00064
00075 std::vector<Frame> handle_get_build_version(const std::string& param, OperationType operationType) {
00076     std::vector<Frame> frames;
00077     std::string error_msg;
00078
00079     if (!param.empty()) {
00080         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00081         frames.push_back(frame_build(OperationType::ERR, 1, 1, error_msg));
00082         return frames;
00083     }
00084
00085     if (operationType == OperationType::GET) {
00086         frames.push_back(frame_build(OperationType::VAL, 1, 1, std::to_string(BUILD_NUMBER)));
00087         return frames;
00088     }
00089
00090     error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00091     frames.push_back(frame_build(OperationType::ERR, 1, 1, error_msg));
00092     return frames;
00093 }
00094
00095
00117 std::vector<Frame> handle_verbosity(const std::string& param, OperationType operationType) {
00118     std::vector<Frame> frames;
00119     std::string error_msg;
00120
00121     if (operationType == OperationType::GET && param.empty()) {
00122         VerbosityLevel current_level = SystemStateManager::get_instance().get_uart_verbosity();
00123         uart_print("GET_VERBOSITY_ " + std::to_string(static_cast<int>(current_level)),
00124                    VerbosityLevel::INFO);
00125         frames.push_back(frame_build(OperationType::VAL, 1, 8,
00126                                std::to_string(static_cast<int>(current_level))));
00127         return frames;
00128     }
00129
00130     try {
00131         int level = std::stoi(param);
00132         if (level < 0 || level > 5) {
00133             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00134             frames.push_back(frame_build(OperationType::ERR, 1, 8, error_msg));
00135             return frames;
00136         }
00137         SystemStateManager::get_instance().set_uart_verbosity(static_cast<VerbosityLevel>(level));
00138         frames.push_back(frame_build(OperationType::RES, 1, 8, "LEVEL SET"));
00139         return frames;
00140     } catch (...) {
00141         error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
00142         frames.push_back(frame_build(OperationType::ERR, 1, 8, error_msg));
00143         return frames;
00144     }
00145 }
00146
00157 std::vector<Frame> handle_enter_bootloader_mode(const std::string& param, OperationType operationType)
{
00158     std::vector<Frame> frames;
00159     std::string error_msg;
00160
00161     if (param != "USB") {
00162         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00163         frames.push_back(frame_build(OperationType::ERR, 1, 9, error_msg));
00164         return frames;
00165     }
00166
00167     if (operationType != OperationType::SET) {
00168         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00169         frames.push_back(frame_build(OperationType::ERR, 1, 9, error_msg));
00170         return frames;
00171     }
00172
00173     frames.push_back(frame_build(OperationType::RES, 1, 9, "REBOOT BOOTSEL"));
00174
00175     SystemStateManager::get_instance().set_bootloader_reset_pending(true);
00176
00177     return frames;
00178 }
00179

```

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

```
#include "communication.h"
#include "event_manager.h"
#include <sstream>
Include dependency graph for event_commands.cpp:
```



Functions

- std::vector< Frame > handle_get_last_events (const std::string ¶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.

8.18 event_commands.cpp

Go to the documentation of this file.

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

```

00069     const EventLog& event = eventManager.get_event(event_index);
00070
00071     ss << std::setw(4) << event.id
00072     << std::setw(8) << event.timestamp
00073     << std::setw(2) << static_cast<int>(event.group)
00074     << std::setw(2) << static_cast<int>(event.event);
00075
00076     if (to_return > 1) ss << "-";
00077
00078     to_return--;
00079     events_in_frame++;
00080 }
00081 frames.push_back(frame_build(OperationType::SEQ, 5, 1, ss.str()));
00082 }
00083 frames.push_back(frame_build(OperationType::VAL, 5, 1, "SEQ_DONE"));
00084 return frames;
00085 }
00086
00087
00100 std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType) {
00101     std::vector<Frame> frames;
00102     std::string error_msg;
00103
00104     if (operationType != OperationType::GET || !param.empty()) {
00105         error_msg = error_code_to_string(KeyCode::INVALID_OPERATION);
00106         frames.push_back(frame_build(OperationType::ERR, 5, 2, error_msg));
00107         return frames;
00108     }
00109     frames.push_back(frame_build(OperationType::VAL, 5, 2,
00110         std::to_string(eventManager.get_event_count())));
00110     return frames;
00111 } // end of EventCommands group

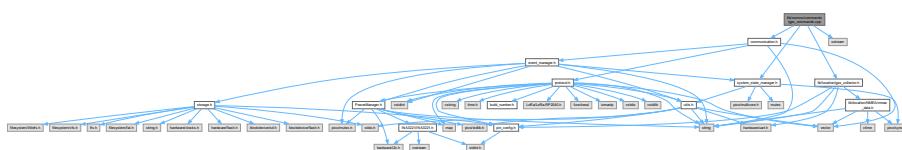
```

8.19 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:

```



Macros

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

Functions

- std::vector< Frame > handle_gps_power_status (const std::string ¶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_rmc_data (const std::string ¶m, OperationType operationType)
Handler for retrieving GPS RMC (Recommended Minimum Navigation) data.
 - std::vector< Frame > handle_get_gga_data (const std::string ¶m, OperationType operationType)
Handler for retrieving GPS GGA (Global Positioning System Fix Data) data.

8.19.1 Macro Definition Documentation

8.19.1.1 GPS_GROUP

```
#define GPS_GROUP 7
```

Definition at line 6 of file [gps_commands.cpp](#).

8.19.1.2 POWER_STATUS_COMMAND

```
#define POWER_STATUS_COMMAND 1
```

Definition at line 7 of file [gps_commands.cpp](#).

8.19.1.3 PASSTHROUGH_COMMAND

```
#define PASSTHROUGH_COMMAND 2
```

Definition at line 8 of file [gps_commands.cpp](#).

8.19.1.4 RMC_DATA_COMMAND

```
#define RMC_DATA_COMMAND 3
```

Definition at line 9 of file [gps_commands.cpp](#).

8.19.1.5 GGA_DATA_COMMAND

```
#define GGA_DATA_COMMAND 4
```

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

8.20 gps_commands.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002 #include "lib/location/gps_collector.h"
00003 #include <sstream>
00004 #include "system_state_manager.h"
00005
00006 #define GPS_GROUP 7
00007 #define POWER_STATUS_COMMAND 1
00008 #define PASSTHROUGH_COMMAND 2
00009 #define RMC_DATA_COMMAND 3
0010 #define GGA_DATA_COMMAND 4
0011
0012
0013 std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType) {
0014     std::vector<Frame> frames;
0015     std::string error_str;
0016
0017     if (operationType == OperationType::SET) {
0018         if (param.empty()) {
0019             error_str = error_code_to_string(ErrorCode::PARAM_REQUIRED);
0020             frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
0021                 error_str));
0022         }
0023
0024         try {
0025             int power_status = std::stoi(param);
0026             if (power_status != 0 && power_status != 1) {
0027                 error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
0028                 frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
0029                     error_str));
0030             }
0031             return frames;
0032         }
0033         gpio_put(GPS_POWER_ENABLE_PIN, power_status);
0034         EventEmitter::emit(EventGroup::GPS, power_status ? GPSEvent::POWER_ON :
0035             GPSEvent::POWER_OFF);
0036         frames.push_back(frame_build(OperationType::RES, GPS_GROUP, POWER_STATUS_COMMAND,
0037             std::to_string(power_status)));
0038         return frames;
0039     } catch (...) {
0040         error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
0041         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
0042             error_str));
0043     }
0044 }
0045
0046 // GET operation
0047 if (!param.empty()) {
0048     error_str = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
0049     frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
0050 }
0051
0052 bool power_status = gpio_get(GPS_POWER_ENABLE_PIN);
0053 frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, POWER_STATUS_COMMAND,
0054     std::to_string(power_status)));
0055 return frames;
0056
0057
0058 std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType
0059     operationType) {
0060     std::vector<Frame> frames;
0061     std::string error_str;
0062
0063     if (!(operationType == OperationType::SET)) {
0064         error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
0065         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
0066     }
0067
0068     // Parse and validate timeout parameter
0069     uint32_t timeout_ms;
0070     try {
0071         timeout_ms = param.empty() ? 60000u : std::stoul(param) * 1000;
0072     } catch (...) {
0073         error_str = error_code_to_string(ErrorCode::INVALID_VALUE);
0074         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
0075     }
0076
0077     // Setup UART parameters and exit sequence

```

```

00111 const std::string EXIT_SEQUENCE = "##EXIT##";
00112 std::string input_buffer;
00113 bool exit_requested = false;
00114 SystemStateManager::get_instance().set_gps_collection_paused(true);
00115 sleep_ms(100);
00116
00117 uint32_t original_baud_rate = DEBUG_UART_BAUD_RATE;
00118 uint32_t gps_baud_rate = GPS_UART_BAUD_RATE;
00119 uint32_t start_time = to_ms_since_boot(get_absolute_time());
00120
00121 EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_START);
00122
00123 std::string message = "Entering GPS Serial Pass-Through Mode @" +
00124             std::to_string(gps_baud_rate) + " for " +
00125             std::to_string(timeout_ms/1000) + "s\r\n" +
00126             "Send " + EXIT_SEQUENCE + " to exit";
00127 uart_print(message, VerbosityLevel::INFO);
00128 sleep_ms(10);
00129
00130 uart_set_baudrate(DEBUG_UART_PORT, gps_baud_rate);
00131
00132 while (!exit_requested) {
00133     while (uart_is_readable(DEBUG_UART_PORT)) {
00134         char ch = uart_getc(DEBUG_UART_PORT);
00135
00136         input_buffer += ch;
00137         if (input_buffer.length() > EXIT_SEQUENCE.length()) {
00138             input_buffer = input_buffer.substr(1);
00139         }
00140
00141         if (input_buffer == EXIT_SEQUENCE) {
00142             exit_requested = true;
00143             break;
00144         }
00145
00146         if (input_buffer != EXIT_SEQUENCE.substr(0, input_buffer.length())) {
00147             uart_write_blocking(GPS_UART_PORT,
00148                 reinterpret_cast<const uint8_t*>(&ch), 1);
00149         }
00150     }
00151
00152     while (uart_is_readable(GPS_UART_PORT)) {
00153         char gps_byte = uart_getc(GPS_UART_PORT);
00154         uart_write_blocking(DEBUG_UART_PORT,
00155             reinterpret_cast<const uint8_t*>(&gps_byte), 1);
00156     }
00157
00158     if (to_ms_since_boot(get_absolute_time()) - start_time >= timeout_ms) {
00159         break;
00160     }
00161 }
00162
00163 uart_set_baudrate(DEBUG_UART_PORT, original_baud_rate);
00164 sleep_ms(50);
00165
00166 SystemStateManager::get_instance().set_gps_collection_paused(false);
00167
00168 EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_END);
00169
00170 std::string exit_reason = exit_requested ? "USER_EXIT" : "TIMEOUT";
00171 std::string response = "GPS UART BRIDGE EXIT: " + exit_reason;
00172 uart_print(response, VerbosityLevel::INFO);
00173
00174 frames.push_back(frame_build(OperationType::RES, GPS_GROUP, PASSTHROUGH_COMMAND, response));
00175
00176 return frames;
00177 }
00178
00179
00180
00181 std::vector<Frame> handle_get_rmc_data(const std::string& param, OperationType operationType) {
00182     std::vector<Frame> frames;
00183     std::string error_msg;
00184
00185     if (operationType != OperationType::GET) {
00186         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00187         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, RMC_DATA_COMMAND, error_msg));
00188         return frames;
00189     }
00190
00191     if (!param.empty()) {
00192         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00193         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, RMC_DATA_COMMAND, error_msg));
00194         return frames;
00195     }
00196
00197     std::vector<std::string> tokens = nmea_data.get_rmc_tokens();

```

```

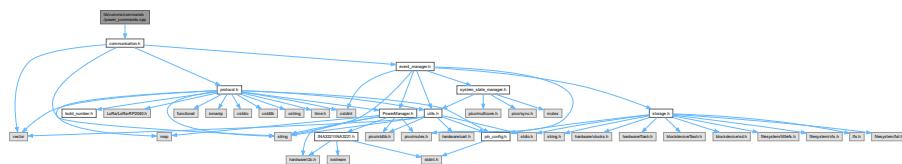
00210     std::stringstream ss;
00211     for (size_t i = 0; i < tokens.size(); ++i) {
00212         ss « tokens[i];
00213         if (i < tokens.size() - 1) {
00214             ss « ",";
00215         }
00216     }
00217     frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, RMC_DATA_COMMAND, ss.str()));
00218     return frames;
00219 }
00220 }
00221
00222
00235 std::vector<Frame> handle_get_gga_data(const std::string& param, OperationType operationType) {
00236     std::vector<Frame> frames;
00237     std::string error_msg;
00238
00239     if (operationType != OperationType::GET) {
00240         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00241         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, GGA_DATA_COMMAND, error_msg));
00242     }
00243     return frames;
00244
00245     if (!param.empty()) {
00246         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00247         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, GGA_DATA_COMMAND, error_msg));
00248     }
00249 }
00250
00251     std::vector<std::string> tokens = nmea_data.get_gga_tokens();
00252     std::stringstream ss;
00253     for (size_t i = 0; i < tokens.size(); ++i) {
00254         ss « tokens[i];
00255         if (i < tokens.size() - 1) {
00256             ss « ",";
00257         }
00258     }
00259
00260     frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, GGA_DATA_COMMAND, ss.str()));
00261     return frames;
00262 } // end of GPSCommands group

```

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

#include "communication.h"

Include dependency graph for power_commands.cpp:



Macros

- #define POWER_GROUP 2
- #define POWER_MANAGER_IDS 0
- #define VOLTAGE_BATTERY 2
- #define VOLTAGE_MAIN 3
- #define CHARGE_USB 4
- #define CHARGE_SOLAR 5
- #define CHARGE_TOTAL 6
- #define DRAW_TOTAL 7

Functions

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

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

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

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

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

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

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

Handler for getting system current draw.

8.21.1 Macro Definition Documentation

8.21.1.1 POWER_GROUP

```
#define POWER_GROUP 2
```

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

8.21.1.2 POWER_MANAGER_IDS

```
#define POWER_MANAGER_IDS 0
```

Definition at line 4 of file [power_commands.cpp](#).

8.21.1.3 VOLTAGE_BATTERY

```
#define VOLTAGE_BATTERY 2
```

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

8.21.1.4 VOLTAGE_MAIN

```
#define VOLTAGE_MAIN 3
```

Definition at line 6 of file [power_commands.cpp](#).

8.21.1.5 CHARGE_USB

```
#define CHARGE_USB 4
```

Definition at line 7 of file [power_commands.cpp](#).

8.21.1.6 CHARGE_SOLAR

```
#define CHARGE_SOLAR 5
```

Definition at line 8 of file [power_commands.cpp](#).

8.21.1.7 CHARGE_TOTAL

```
#define CHARGE_TOTAL 6
```

Definition at line 9 of file [power_commands.cpp](#).

8.21.1.8 DRAW_TOTAL

```
#define DRAW_TOTAL 7
```

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

8.22 power_commands.cpp

[Go to the documentation of this file.](#)

```
00001 #include "communication.h"
00002
00003 #define POWER_GROUP 2
00004 #define POWER_MANAGER_IDS 0
00005 #define VOLTAGE_BATTERY 2
00006 #define VOLTAGE_MAIN 3
00007 #define CHARGE_USB 4
00008 #define CHARGE_SOLAR 5
00009 #define CHARGE_TOTAL 6
00010 #define DRAW_TOTAL 7
00011
00012
00030 std::vector<Frame> handle_get_power_manager_ids(const std::string& param, OperationType operationType)
{
00031     std::vector<Frame> frames;
00032     std::string error_msg;
00033
00034     if (!param.empty()) {
00035         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00036         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, POWER_MANAGER_IDS, error_msg));
00037         return frames;
00038     }
00039
00040     if (!(operationType == OperationType::GET)) {
00041         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00042         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, POWER_MANAGER_IDS, error_msg));
00043         return frames;
00044     }
00045
00046     extern PowerManager powerManager;
00047     std::string power_manager_ids = powerManager.read_device_ids();
00048     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, POWER_MANAGER_IDS,
00049     power_manager_ids));
00049     return frames;
```

```

00050 }
00051
00064 std::vector<Frame> handle_get_voltage_battery(const std::string& param, OperationType operationType) {
00065     std::vector<Frame> frames;
00066     std::string error_msg;
00067
00068     if (!param.empty()) {
00069         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00070         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_BATTERY, error_msg));
00071         return frames;
00072     }
00073
00074     if (!(operationType == OperationType::GET)) {
00075         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00076         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_BATTERY, error_msg));
00077         return frames;
00078     }
00079
00080     extern PowerManager powerManager;
00081     float voltage = powerManager.get_voltage_battery();
00082     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, VOLTAGE_BATTERY,
00083         std::to_string(voltage), ValueUnit::VOLT));
00084     return frames;
00085
00098 std::vector<Frame> handle_get_voltage_5v(const std::string& param, OperationType operationType) {
00099     std::vector<Frame> frames;
00100     std::string error_msg;
00101
00102     if (!param.empty()) {
00103         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00104         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_MAIN, error_msg));
00105         return frames;
00106     }
00107
00108     if (!(operationType == OperationType::GET)) {
00109         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00110         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_MAIN, error_msg));
00111         return frames;
00112     }
00113
00114     extern PowerManager powerManager;
00115     float voltage = powerManager.get_voltage_5v();
00116     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, VOLTAGE_MAIN,
00117         std::to_string(voltage), ValueUnit::VOLT));
00118     return frames;
00119
00132 std::vector<Frame> handle_get_current_charge_usb(const std::string& param, OperationType
00133     operationType) {
00134     std::vector<Frame> frames;
00135     std::string error_msg;
00136
00137     if (!param.empty()) {
00138         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00139         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_USB, error_msg));
00140         return frames;
00141
00142     if (!(operationType == OperationType::GET)) {
00143         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00144         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_USB, error_msg));
00145         return frames;
00146     }
00147
00148     extern PowerManager powerManager;
00149     float charge_current = powerManager.get_current_charge_usb();
00150     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_USB,
00151         std::to_string(charge_current), ValueUnit::MILIAMP));
00152     return frames;
00153
00166 std::vector<Frame> handle_get_current_charge_solar(const std::string& param, OperationType
00167     operationType) {
00168     std::vector<Frame> frames;
00169     std::string error_msg;
00170
00171     if (!param.empty()) {
00172         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00173         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_SOLAR, error_msg));
00174         return frames;
00175
00176     if (!(operationType == OperationType::GET)) {
00177         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00178         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_SOLAR, error_msg));
00179         return frames;

```

```

00180     }
00181
00182     extern PowerManager powerManager;
00183     float charge_current = powerManager.get_current_charge_solar();
00184     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_SOLAR,
00185         std::to_string(charge_current), ValueUnit::MILIAMP));
00186     return frames;
00187 }
00188
00189 std::vector<Frame> handle_get_current_charge_total(const std::string& param, OperationType
00190 operationType) {
00191     std::vector<Frame> frames;
00192     std::string error_msg;
00193
00194     if (!param.empty()) {
00195         error_msg = error_code_to_string(KeyCode::PARAM_UNNECESSARY);
00196         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_TOTAL, error_msg));
00197         return frames;
00198     }
00199
00200     if (!(operationType == OperationType::GET)) {
00201         error_msg = error_code_to_string(KeyCode::INVALID_OPERATION);
00202         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_TOTAL, error_msg));
00203         return frames;
00204     }
00205
00206     extern PowerManager powerManager;
00207     float charge_current = powerManager.get_current_charge_total();
00208     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_TOTAL,
00209         std::to_string(charge_current), ValueUnit::MILIAMP));
00210     return frames;
00211 }
00212
00213 std::vector<Frame> handle_get_current_draw(const std::string& param, OperationType operationType) {
00214     std::vector<Frame> frames;
00215     std::string error_msg;
00216
00217     if (!param.empty()) {
00218         error_msg = error_code_to_string(KeyCode::PARAM_UNNECESSARY);
00219         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, DRAW_TOTAL, error_msg));
00220         return frames;
00221     }
00222
00223     if (!(operationType == OperationType::GET)) {
00224         error_msg = error_code_to_string(KeyCode::INVALID_OPERATION);
00225         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, DRAW_TOTAL, error_msg));
00226         return frames;
00227     }
00228
00229     extern PowerManager powerManager;
00230     float current_draw = powerManager.get_current_draw();
00231     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, DRAW_TOTAL,
00232         std::to_string(current_draw), ValueUnit::MILIAMP));
00233     return frames;
00234 } // end of PowerCommands group

```

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

```

#include "communication.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <sstream>
#include "commands.h"
Include dependency graph for sensor_commands.cpp:

```



Macros

- #define SENSOR_GROUP 4
- #define SENSOR_READ 0
- #define SENSOR_CONFIGURE 1

Functions

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

8.23.1 Macro Definition Documentation

8.23.1.1 SENSOR_GROUP

```
#define SENSOR_GROUP 4
```

Definition at line 8 of file [sensor_commands.cpp](#).

8.23.1.2 SENSOR_READ

```
#define SENSOR_READ 0
```

Definition at line 9 of file [sensor_commands.cpp](#).

8.23.1.3 SENSOR_CONFIGURE

```
#define SENSOR_CONFIGURE 1
```

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

8.24 sensor_commands.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002 #include "ISensor.h"
00003 #include <vector>
00004 #include <string>
00005 #include <iostream>
00006 #include "commands.h"
00007
00008 #define SENSOR_GROUP 4
00009 #define SENSOR_READ 0
0010 #define SENSOR_CONFIGURE 1
0011
0017
0036 std::vector<Frame> handle_get_sensor_data(const std::string& param, OperationType operationType) {
0037     std::vector<Frame> frames;
0038     std::string error_msg;
0039
0040     if (param.empty()) {
0041         error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
0042         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
0043         return frames;
0044     }
0045
0046     if (operationType != OperationType::GET) {
0047         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
0048         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
0049         return frames;
0050     }
0051
0052     // Parse sensor type and data type from param
0053     std::string sensor_type_str;
0054     std::string data_type_str;
0055
0056     size_t dash_pos = param.find('-');
0057     if (dash_pos != std::string::npos) {
0058         sensor_type_str = param.substr(0, dash_pos);
0059         data_type_str = param.substr(dash_pos + 1);
0060     } else {
0061         sensor_type_str = param;
0062     }
0063
0064     // Convert sensor_type_str to SensorType
0065     SensorType sensor_type;
0066     if (sensor_type_str == "light") {
0067         sensor_type = SensorType::LIGHT;
0068     } else if (sensor_type_str == "environment") {
0069         sensor_type = SensorType::ENVIRONMENT;
0070     } else if (sensor_type_str == "magnetometer") {
0071         sensor_type = SensorType::MAGNETOMETER;
0072     } else if (sensor_type_str == "imu") {
0073         sensor_type = SensorType::IMU;
0074     } else {
0075         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid sensor type";
0076         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
0077         return frames;
0078     }
0079
0080     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
0081
0082     // If data type is specified, read that specific data
0083     if (!data_type_str.empty()) {
0084         SensorDataTypeIdentifier data_type;
0085
0086         // Map string to SensorDataTypeIdentifier
0087         if (data_type_str == "light_level") {
0088             data_type = SensorDataTypeIdentifier::LIGHT_LEVEL;
0089         } else if (data_type_str == "temperature") {
0090             data_type = SensorDataTypeIdentifier::TEMPERATURE;
0091         } else if (data_type_str == "pressure") {
0092             data_type = SensorDataTypeIdentifier::PRESSURE;
0093         } else if (data_type_str == "humidity") {
0094             data_type = SensorDataTypeIdentifier::HUMIDITY;
0095         } else if (data_type_str == "mag_field_x") {
0096             data_type = SensorDataTypeIdentifier::MAG_FIELD_X;
0097         } else if (data_type_str == "mag_field_y") {
0098             data_type = SensorDataTypeIdentifier::MAG_FIELD_Y;
0099         } else if (data_type_str == "mag_field_z") {
0100             data_type = SensorDataTypeIdentifier::MAG_FIELD_Z;
0101         } else if (data_type_str == "gyro_x") {
0102             data_type = SensorDataTypeIdentifier::GYRO_X;
0103         } else if (data_type_str == "gyro_y") {
0104             data_type = SensorDataTypeIdentifier::GYRO_Y;
0105         } else if (data_type_str == "gyro_z") {
0106

```

```

00106     data_type = SensorDataTypeIdentifier::GYRO_Z;
00107 } else if (data_type_str == "accel_x") {
00108     data_type = SensorDataTypeIdentifier::ACCEL_X;
00109 } else if (data_type_str == "accel_y") {
00110     data_type = SensorDataTypeIdentifier::ACCEL_Y;
00111 } else if (data_type_str == "accel_z") {
00112     data_type = SensorDataTypeIdentifier::ACCEL_Z;
00113 } else {
00114     error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid data type";
00115     frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
00116     return frames;
00117 }
00118
00119     float value = sensor_wrapper.sensor_read_data(sensor_type, data_type);
00120     std::stringstream ss;
00121     ss << value;
00122     frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, SENSOR_READ, ss.str()));
00123 }
00124 // If only sensor type is specified, read all relevant data for that sensor type
00125 else {
00126     std::vector<SensorDataTypeIdentifier> data_types;
00127
00128     switch (sensor_type) {
00129         case SensorType::LIGHT:
00130             data_types = {SensorDataTypeIdentifier::LIGHT_LEVEL};
00131             break;
00132         case SensorType::ENVIRONMENT:
00133             data_types = {
00134                 SensorDataTypeIdentifier::TEMPERATURE,
00135                 SensorDataTypeIdentifier::PRESSURE,
00136                 SensorDataTypeIdentifier::HUMIDITY
00137             };
00138             break;
00139         case SensorType::MAGNETOMETER:
00140             data_types = {
00141                 SensorDataTypeIdentifier::MAG_FIELD_X,
00142                 SensorDataTypeIdentifier::MAG_FIELD_Y,
00143                 SensorDataTypeIdentifier::MAG_FIELD_Z
00144             };
00145             break;
00146         case SensorType::IMU:
00147             data_types = {
00148                 SensorDataTypeIdentifier::GYRO_X,
00149                 SensorDataTypeIdentifier::GYRO_Y,
00150                 SensorDataTypeIdentifier::GYRO_Z,
00151                 SensorDataTypeIdentifier::ACCEL_X,
00152                 SensorDataTypeIdentifier::ACCEL_Y,
00153                 SensorDataTypeIdentifier::ACCEL_Z
00154             };
00155             break;
00156     }
00157
00158     std::stringstream combined_values;
00159     std::vector<std::string> data_type_names;
00160     std::vector<float> values;
00161
00162 // Get names for the data types and store the values
00163 for (SensorDataTypeIdentifier data_type : data_types) {
00164     switch (data_type) {
00165         case SensorDataTypeIdentifier::LIGHT_LEVEL:
00166             data_type_names.push_back("light_level");
00167             break;
00168         case SensorDataTypeIdentifier::TEMPERATURE:
00169             data_type_names.push_back("temperature");
00170             break;
00171         case SensorDataTypeIdentifier::PRESSURE:
00172             data_type_names.push_back("pressure");
00173             break;
00174         case SensorDataTypeIdentifier::HUMIDITY:
00175             data_type_names.push_back("humidity");
00176             break;
00177         case SensorDataTypeIdentifier::MAG_FIELD_X:
00178             data_type_names.push_back("mag_field_x");
00179             break;
00180         case SensorDataTypeIdentifier::MAG_FIELD_Y:
00181             data_type_names.push_back("mag_field_y");
00182             break;
00183         case SensorDataTypeIdentifier::MAG_FIELD_Z:
00184             data_type_names.push_back("mag_field_z");
00185             break;
00186         case SensorDataTypeIdentifier::GYRO_X:
00187             data_type_names.push_back("gyro_x");
00188             break;
00189         case SensorDataTypeIdentifier::GYRO_Y:
00190             data_type_names.push_back("gyro_y");
00191             break;
00192         case SensorDataTypeIdentifier::GYRO_Z:
00193             data_type_names.push_back("gyro_z");
00194     }
00195 }
```

```

00193             data_type_names.push_back("gyro_z");
00194             break;
00195         case SensorDataTypeIdentifier::ACCEL_X:
00196             data_type_names.push_back("accel_x");
00197             break;
00198         case SensorDataTypeIdentifier::ACCEL_Y:
00199             data_type_names.push_back("accel_y");
00200             break;
00201         case SensorDataTypeIdentifier::ACCEL_Z:
00202             data_type_names.push_back("accel_z");
00203             break;
00204     }
00205
00206     float value = sensor_wrapper.sensor_read_data(sensor_type, data_type);
00207     values.push_back(value);
00208 }
00209
00210 // Format output as key-value pairs
00211 for (size_t i = 0; i < data_type_names.size(); i++) {
00212     if (i > 0) combined_values « "|";
00213     combined_values « data_type_names[i] « ":" « values[i];
00214 }
00215
00216 frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, SENSOR_READ,
combined_values.str()));
00217 }
00218
00219 return frames;
00220 }
00221
002236 std::vector<Frame> handle_sensor_config(const std::string& param, OperationType operationType) {
002237     std::vector<Frame> frames;
002238     std::string error_msg;
002239
002240     if (param.empty()) {
002241         error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
002242         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
002243         return frames;
002244     }
002245
002246     if (operationType != OperationType::SET) {
002247         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
002248         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
002249         return frames;
002250     }
002251
002252     // Parse sensor type and configuration from param
002253     size_t semicolon_pos = param.find(';");
002254     if (semicolon_pos == std::string::npos) {
002255         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Format should be
sensor_type;config_params";
002256         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
002257         return frames;
002258     }
002259
002260     std::string sensor_type_str = param.substr(0, semicolon_pos);
002261     std::string config_str = param.substr(semicolon_pos + 1);
002262
002263     // Convert sensor_type_str to SensorType
002264     SensorType sensor_type;
002265     if (sensor_type_str == "light") {
002266         sensor_type = SensorType::LIGHT;
002267     } else if (sensor_type_str == "environment") {
002268         sensor_type = SensorType::ENVIRONMENT;
002269     } else if (sensor_type_str == "magnetometer") {
002270         sensor_type = SensorType::MAGNETOMETER;
002271     } else if (sensor_type_str == "imu") {
002272         sensor_type = SensorType::IMU;
002273     } else {
002274         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid sensor type";
002275         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
002276         return frames;
002277     }
002278
002279     // Parse configuration parameters
002280     std::map<std::string, std::string> config_map;
002281     std::stringstream ss(config_str);
002282     std::string config_pair;
002283
002284     while (std::getline(ss, config_pair, '|')) {
002285         size_t colon_pos = config_pair.find(':');
002286         if (colon_pos != std::string::npos) {
002287             std::string key = config_pair.substr(0, colon_pos);
002288             std::string value = config_pair.substr(colon_pos + 1);
002289             config_map[key] = value;
002290         }
002291     }

```

```

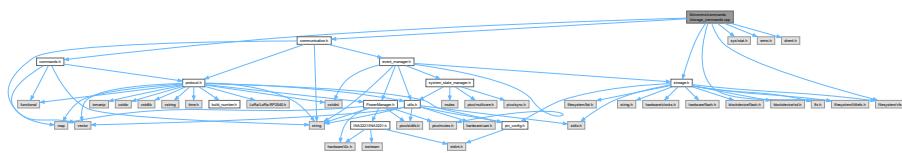
00292
00293 // Apply configuration
00294 SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00295 bool success = sensor_wrapper.sensor_configure(sensor_type, config_map);
00296
00297 if (success) {
00298     frames.push_back(frame_build(OperationType::RES, SENSOR_GROUP, SENSOR_CONFIGURE,
00299     "Configuration successful"));
00300 } else {
00301     error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET) + ": Failed to configure sensor";
00302     frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
00303 }
00304 return frames;
00305 }
00306
00307
00320 std::vector<Frame> handle_get_sensor_list(const std::string& param, OperationType operationType) {
00321     std::vector<Frame> frames;
00322     std::string error_msg;
00323
00324     if (operationType != OperationType::GET)
00325         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00326     frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, 2, error_msg));
00327     return frames;
00328 }
00329
00330 // Get the singleton instance
00331 SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00332
00333 // Get list of available sensor types
00334 std::vector<std::pair<SensorType, uint8_t>> available_sensors =
00335     sensor_wrapper.get_available_sensors();
00336
00337 if (available_sensors.empty()) {
00338     frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, 2, "No sensors available"));
00339     return frames;
00340 }
00341 std::stringstream sensor_list;
00342 bool first = true;
00343
00344 for (const auto& sensor_info : available_sensors) {
00345     if (!first) {
00346         sensor_list << "|";
00347     }
00348
00349     // Format: sensor_type:address (in hex)
00350     std::stringstream addr_hex;
00351     addr_hex << std::hex << static_cast<int>(sensor_info.second);
00352
00353     switch (sensor_info.first) {
00354         case SensorType::LIGHT:
00355             sensor_list << "light:0x" << addr_hex.str();
00356             break;
00357         case SensorType::ENVIRONMENT:
00358             sensor_list << "environment:0x" << addr_hex.str();
00359             break;
00360         case SensorType::MAGNETOMETER:
00361             sensor_list << "magnetometer:0x" << addr_hex.str();
00362             break;
00363         case SensorType::IMU:
00364             sensor_list << "imu:0x" << addr_hex.str();
00365             break;
00366         default:
00367             sensor_list << "unknown:0x" << addr_hex.str();
00368             break;
00369     }
00370
00371     first = false;
00372 }
00373
00374 frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, 2, sensor_list.str()));
00375 return frames;
00376 }

```

8.25 lib/comms/commands/storage_commands.cpp File Reference

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

```
#include "storage.h"
#include "filesystem/vfs.h"
#include "filesystem/littlefs.h"
#include <sys/stat.h>
#include <errno.h>
#include "dirent.h"
Include dependency graph for storage_commands.cpp:
```



Macros

- #define STORAGE_GROUP 6
- #define LIST_FILES_COMMAND 0
- #define MOUNT_COMMAND 4

Functions

- std::vector< Frame > handle_list_files (const std::string ¶m, OperationType operationType)
Handles the list files command.
- std::vector< Frame > handle_mount (const std::string ¶m, OperationType operationType)
Handles the SD card mount/unmount command.

8.25.1 Macro Definition Documentation

8.25.1.1 STORAGE_GROUP

```
#define STORAGE_GROUP 6
```

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

8.25.1.2 LIST_FILES_COMMAND

```
#define LIST_FILES_COMMAND 0
```

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

8.25.1.3 MOUNT_COMMAND

```
#define MOUNT_COMMAND 4
```

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

8.26 storage_commands.cpp

[Go to the documentation of this file.](#)

```

00001 #include "commands.h"
00002 #include "communication.h"
00003 #include "storage.h"
00004 #include "filesystem/vfs.h"
00005 #include "filesystem/littlefs.h"
00006 #include <sys/stat.h>
00007 #include <errno.h>
00008 #include "dirent.h"
00009
00010 #define STORAGE_GROUP 6
00011
00012 #define LIST_FILES_COMMAND 0
00013 #define MOUNT_COMMAND 4
00019
00037 std::vector<Frame> handle_list_files(const std::string& param, OperationType operationType) {
00038     std::vector<Frame> frames;
00039     std::string error_msg;
00040
00041     if (operationType != OperationType::GET) {
00042         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00043         frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND,
00044             error_msg));
00045         return frames;
00046     }
00047
00048     DIR* dir;
00049     struct dirent* ent;
00050     int file_count = 0; // Counter for the number of files
00051     if ((dir = opendir(".")) != NULL) {
00052         // First, count the number of files
00053         while ((ent = readdir(dir)) != NULL) {
00054             const char* filename = ent->d_name;
00055             if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
00056                 continue;
00057             }
00058             file_count++;
00059         }
00060         closedir(dir);
00061
00062         // Send the number of files
00063         frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, LIST_FILES_COMMAND,
00064             std::to_string(file_count)));
00065
00066         // Open the directory again to read file information
00067         dir = opendir("/");
00068         if (dir != NULL) {
00069             while ((ent = readdir(dir)) != NULL) {
00070                 const char* filename = ent->d_name;
00071
00072                 // Skip "." and ".." directories
00073                 if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
00074                     continue;
00075
00076                 // Get file size
00077                 char filepath[256];
00078                 snprintf(filepath, sizeof(filepath), "/%s", filename);
00079
00080                 FILE* file = fopen(filepath, "rb");
00081                 size_t file_size = 0;
00082
00083                 if (file != NULL) {
00084                     fseek(file, 0, SEEK_END);
00085                     file_size = ftell(file);
00086                     fclose(file);
00087                 }
00088
00089                 // Create and send frame with filename and size
00090                 char file_info[512];
00091                 snprintf(file_info, sizeof(file_info), "%s:%zu", filename, file_size);
00092                 frames.push_back(frame_build(OperationType::SEQ, STORAGE_GROUP, LIST_FILES_COMMAND,
00093                     file_info));
00094
00095                 closedir(dir);
00096                 frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, LIST_FILES_COMMAND,
00097                     "SEQ_DONE"));
00098
00099             }
00100         }
00101     }
00102     return frames;
00103 } else {
00104     error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
00105     frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND,
00106         error_msg));
00107 }
00108
00109 return frames;

```

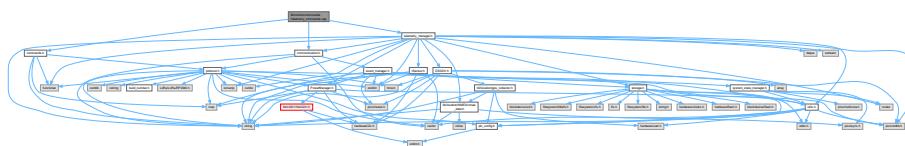
```

00100         }
00101     } else {
00102         frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND,
00103                           error_msg));
00104     }
00105 }
00106
00123 std::vector<Frame> handle_mount(const std::string& param, OperationType operationType) {
00124     std::vector<Frame> frames;
00125     std::string error_msg;
00126
00127     if (operationType == OperationType::GET) {
00128         bool state = SystemStateManager::get_instance().is_sd_card_mounted();
00129
00130         frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, MOUNT_COMMAND,
00131                           std::to_string(state)));
00132         return frames;
00133     } else if (operationType == OperationType::SET) {
00134         if (param == "1") {
00135             if (fs_init()) {
00136                 frames.push_back(frame_build(OperationType::RES, STORAGE_GROUP, MOUNT_COMMAND,
00137                               "SD_MOUNT_OK"));
00138                 return frames;
00139             } else {
00140                 error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
00141                 frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND,
00142                               error_msg));
00143                 return frames;
00144             }
00145             if (fs_unmount("/") == 0) {
00146                 if (SystemStateManager::get_instance().is_sd_card_mounted()) {
00147                     frames.push_back(frame_build(OperationType::RES, STORAGE_GROUP, MOUNT_COMMAND,
00148                               "SD_UNMOUNT_OK"));
00149                     return frames;
00150                 } else {
00151                     error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
00152                     frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND,
00153                               error_msg));
00154                     return frames;
00155                 }
00156             }
00157         }
00158     } else {
00159         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00160         frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND,
00161                           error_msg));
00162     }
00163 } // StorageCommands

```

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

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



Macros

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

Functions

- std::vector< Frame > handle_get_last_telemetry_record (const std::string ¶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

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

8.27.1 Macro Definition Documentation

8.27.1.1 TELEMTRY_GROUP

```
#define TELEMETRY_GROUP 8
```

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

8.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND

```
#define GET_LAST_TELEMETRY_RECORD_COMMAND 2
```

Definition at line 6 of file [telemetry_commands.cpp](#).

8.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND

```
#define GET_LAST_SENSOR_RECORD_COMMAND 3
```

Definition at line 7 of file [telemetry_commands.cpp](#).

8.27.2 Variable Documentation

8.27.2.1 telemetry_mutex

```
mutex_t telemetry_mutex [extern]
```

Mutex for thread-safe access to the telemetry buffer.

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

8.27.2.2 telemetry_buffer_count

```
size_t telemetry_buffer_count [extern]
```

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

8.27.2.3 telemetry_buffer_write_index

```
size_t telemetry_buffer_write_index [extern]
```

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

8.28 telemetry_commands.cpp

[Go to the documentation of this file.](#)

```
00001 #include "commands.h"
00002 #include "communication.h"
00003 #include "telemetry_manager.h"
00004
00005 #define TELEMTRY_GROUP 8
00006 #define GET_LAST_TELEMETRY_RECORD_COMMAND 2
00007 #define GET_LAST_SENSOR_RECORD_COMMAND 3
00008
00009 extern mutex_t telemetry_mutex;
00010 extern size_t telemetry_buffer_count;
00011 extern size_t telemetry_buffer_write_index;
00012 extern const int TELEMETRY_BUFFER_SIZE;
00013
00014
00015 std::vector<Frame> handle_get_last_telemetry_record(const std::string& param, OperationType
00016 operationType) {
00017     std::vector<Frame> frames;
00018     std::string error_msg;
00019
00020     if (operationType != OperationType::GET) {
00021         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00022         frames.push_back(frame_build(OperationType::ERR, TELEMTRY_GROUP,
00023             GET_LAST_TELEMETRY_RECORD_COMMAND, error_msg));
00024         return frames;
00025     }
00026
00027     std::string csv_data = get_last_telemetry_record_csv();
00028
00029     if (csv_data.empty()) {
00030         error_msg = "NO_DATA";
00031         frames.push_back(frame_build(OperationType::ERR, TELEMTRY_GROUP,
00032             GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00033         return frames;
00034     }
00035     // Create and send the frame with the base64 encoded data
00036     frames.push_back(frame_build(OperationType::VAL, TELEMTRY_GROUP,
00037         GET_LAST_TELEMETRY_RECORD_COMMAND, csv_data));
00038
00039     return frames;
00040 }
00041
00042
00043 std::vector<Frame> handle_get_last_sensor_record(const std::string& param, OperationType
00044 operationType) {
00045     std::vector<Frame> frames;
00046     std::string error_msg;
00047
00048     if (operationType != OperationType::GET) {
00049         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00050         frames.push_back(frame_build(OperationType::ERR, TELEMTRY_GROUP,
00051             GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00052         return frames;
00053     }
00054     // Create and send the frame with the base64 encoded data
00055     frames.push_back(frame_build(OperationType::VAL, TELEMTRY_GROUP,
00056         GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00057
00058     return frames;
00059 }
00060
00061
00062
00063
00064
00065
00066
00067
00068
00069
00070
00071
00072
00073
00074
00075
00076
00077
00078
00079
00080
00081
00082
00083
```

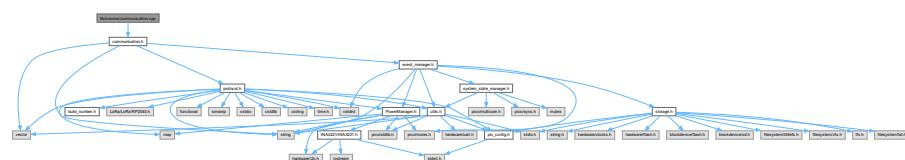
```

00084     error_msg = "NO_DATA";
00085     frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP,
00086                                 GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00087 }
00088
00089 // Create and send the frame with the sensor data
00090 frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND,
00091                               csv_data));
00092
00093 } // TelemetryBufferCommands

```

8.29 lib/comms/communication.cpp File Reference

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



Functions

- bool [initialize_radio \(\)](#)
Initializes the LoRa radio module.
- void [lora_tx_done_callback \(\)](#)

Variables

- string [outgoing](#)
- uint8_t [msgCount](#) = 0
- long [lastSendTime](#) = 0
- long [lastReceiveTime](#) = 0
- long [lastPrintTime](#) = 0
- unsigned long [interval](#) = 0

8.29.1 Function Documentation

8.29.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

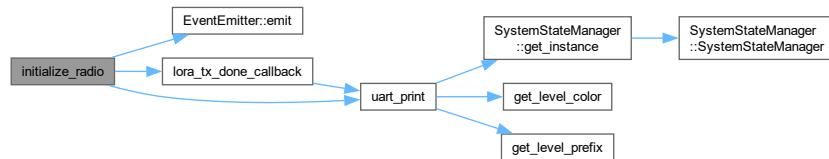
Returns

True if initialization was successful, false otherwise.

Sets the LoRa pins and attempts to begin LoRa communication at a specified frequency. Emits a [CommsEvent::RADIO_INIT](#) event on success or a [CommsEvent::RADIO_ERROR](#) event on failure.

Definition at line 18 of file [communication.cpp](#).

Here is the call graph for this function:



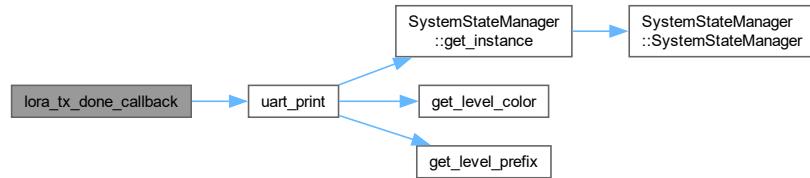
Here is the caller graph for this function:

**8.29.1.2 lora_tx_done_callback()**

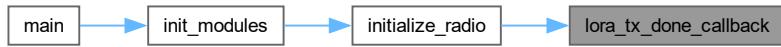
```
void lora_tx_done_callback ()
```

Definition at line 43 of file [communication.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.29.2 Variable Documentation

8.29.2.1 outgoing

```
string outgoing
```

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

8.29.2.2 msgCount

```
uint8_t msgCount = 0
```

Definition at line 4 of file [communication.cpp](#).

8.29.2.3 lastSendTime

```
long lastSendTime = 0
```

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

8.29.2.4 lastReceiveTime

```
long lastReceiveTime = 0
```

Definition at line 6 of file [communication.cpp](#).

8.29.2.5 lastPrintTime

```
long lastPrintTime = 0
```

Definition at line 7 of file [communication.cpp](#).

8.29.2.6 interval

```
unsigned long interval = 0
```

Definition at line 8 of file [communication.cpp](#).

8.30 communication.cpp

[Go to the documentation of this file.](#)

```

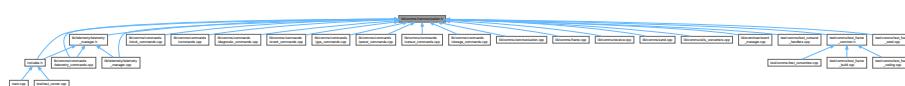
00001 #include "communication.h"
00002
00003 string outgoing;
00004 uint8_t msgCount = 0;
00005 long lastSendTime = 0;
00006 long lastReceiveTime = 0;
00007 long lastPrintTime = 0;
00008 unsigned long interval = 0;
00009
00010
00017 // In initialize_radio() function in communication.cpp
00018 bool initialize_radio() {
00019     LoRa.set_pins(lora_cs_pin, lora_reset_pin, lora_irq_pin);
00020     long frequency = 433E6;
00021     bool init_status = false;
00022     if (!LoRa.begin(frequency))
00023     {
00024         uart_print("LoRa init failed. Check your connections.", VerbosityLevel::WARNING);
00025         init_status = false;
00026     } else {
00027         uart_print("LoRa initialized with frequency " + std::to_string(frequency),
00028             VerbosityLevel::INFO);
00029         // Set up TxDone callback to automatically return to receive mode
00030         LoRa.onTxDone(lora_tx_done_callback);
00031
00032         // Start in receive mode
00033         LoRa.receive(0);
00034
00035         init_status = true;
00036     }
00037
00038     EventEmitter::emit(EventGroup::COMMS, init_status ? CommsEvent::RADIO_INIT :
00039         CommsEvent::RADIO_ERROR);
00040
00041     return init_status;
00042 }
00043 void lora_tx_done_callback() {
00044     uart_print("LoRa transmission complete", VerbosityLevel::DEBUG);
00045     LoRa.receive(0);
00046 }
```

8.31 lib/comms/communication.h File Reference

```
#include <string>
#include <vector>
#include "protocol.h"
#include "event_manager.h"
Include dependency graph for communication.h:
```



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



Functions

- `bool initialize_radio ()`
Initializes the LoRa radio module.
- `void lora_tx_done_callback ()`
- `void on_receive (int packetSize)`
Callback function for handling received LoRa packets.
- `void handle_uart_input ()`
Handles UART input.
- `void send_message (std::string outgoing)`
- `void send_frame_uart (const Frame &frame)`
- `void send_frame_lora (const Frame &frame)`
- `void split_and_send_message (const uint8_t *data, size_t length)`
- `std::vector< Frame > execute_command (uint32_t commandKey, const std::string ¶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.
- `std::string determine_unit (uint8_t group, uint8_t command)`

8.31.1 Function Documentation

8.31.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

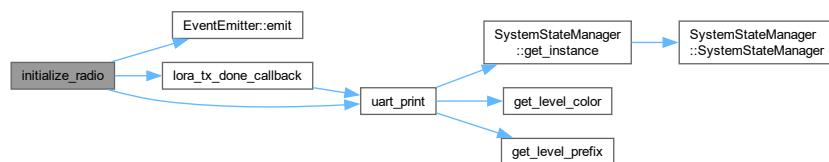
Returns

True if initialization was successful, false otherwise.

Sets the LoRa pins and attempts to begin LoRa communication at a specified frequency. Emits a `CommsEvent::RADIO_INIT` event on success or a `CommsEvent::RADIO_ERROR` event on failure.

Definition at line 18 of file `communication.cpp`.

Here is the call graph for this function:



Here is the caller graph for this function:

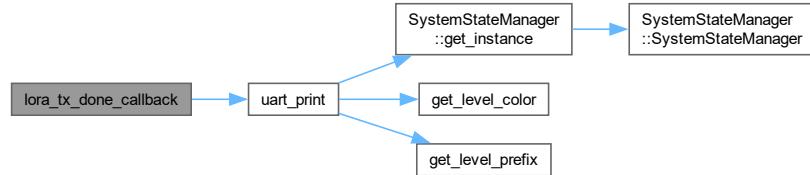


8.31.1.2 lora_tx_done_callback()

```
void lora_tx_done_callback ()
```

Definition at line 43 of file [communication.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.31.1.3 on_receive()

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

Callback function for handling received LoRa packets.

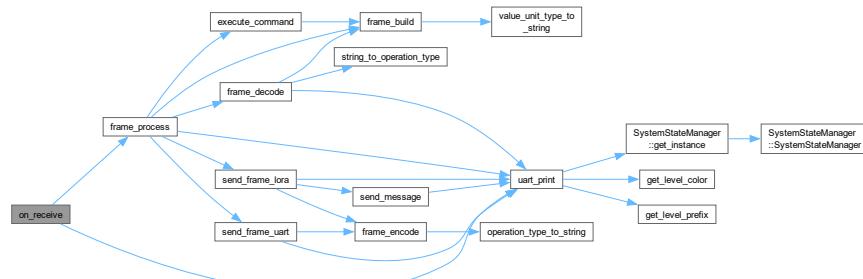
Parameters

<i>packet_size</i>	The size of the received packet.
--------------------	----------------------------------

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



8.31.1.4 handle_uart_input()

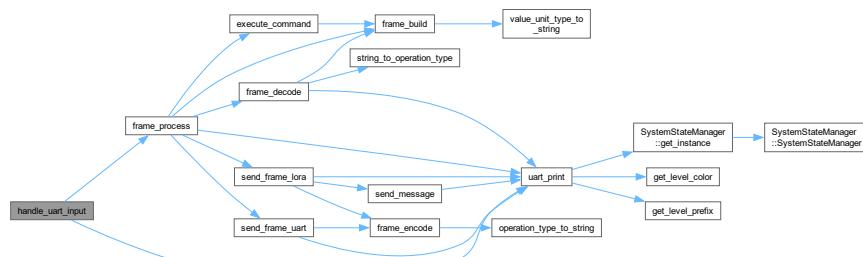
```
void handle_uart_input ()
```

Handles UART input.

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

Definition at line 86 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.31.1.5 send_message()

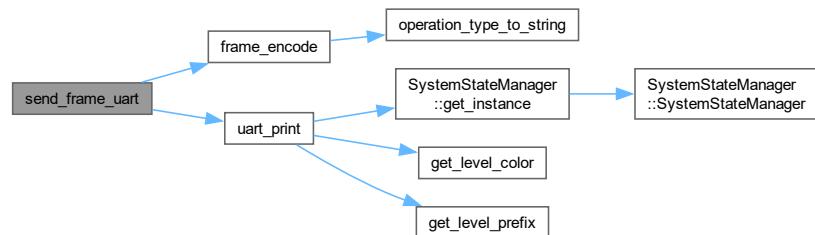
```
void send_message (
    std::string outgoing)
```

8.31.1.6 send_frame_uart()

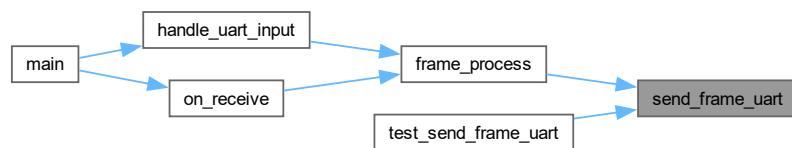
```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 46 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

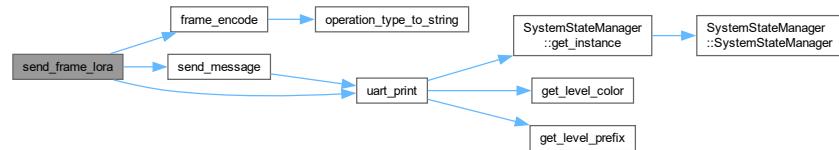


8.31.1.7 send_frame_lora()

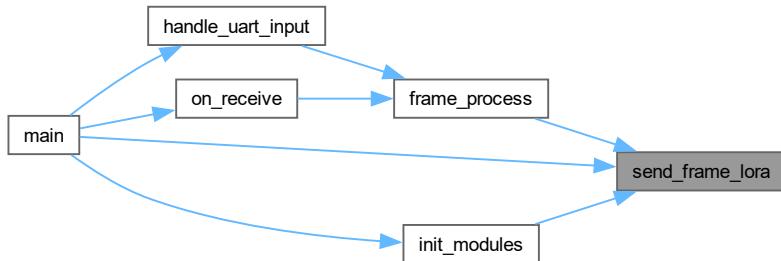
```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 39 of file [send.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.31.1.8 split_and_send_message()

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

8.31.1.9 determine_unit()

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

8.32 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 void split_and_send_message(const uint8_t* data, size_t length);
00018
00019 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
00020 operationType);
00021 void frame_process(const std::string& data, Interface interface);
00022 std::string frame_encode(const Frame& frame);
00023 Frame frame_decode(const std::string& data);
00024 Frame frame_build(OperationType operation, uint8_t group, uint8_t command, const std::string& value,
00025 const ValueUnit unitType = ValueUnit::UNDEFINED);
00026 std::string determine_unit(uint8_t group, uint8_t command);
00027
00028 #endif

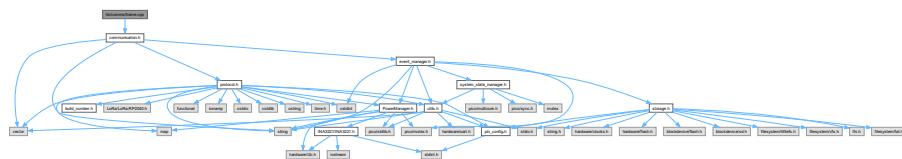
```

8.33 lib/comms/frame.cpp File Reference

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

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

Include dependency graph for frame.cpp:



TypeDefs

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

Functions

- std::string **frame_encode** (const Frame &frame)
Encodes a Frame instance into a string.
- Frame **frame_decode** (const std::string &data)
Decodes a string into a Frame instance.
- void **frame_process** (const std::string &data, Interface interface)
Executes a command based on the command key and the parameter.
- Frame **frame_build** (OperationType operation, uint8_t group, uint8_t command, const std::string &value, const ValueUnit unitType)
Builds a Frame instance based on the execution result, group, command, value, and unit.

Variables

- std::map< uint32_t, CommandHandler > command_handlers
Global map of all command handlers.
- volatile uint16_t eventRegister

8.33.1 Detailed Description

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

Definition in file [frame.cpp](#).

8.33.2 Typedef Documentation

8.33.2.1 CommandHandler

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

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

8.33.3 Variable Documentation

8.33.3.1 eventRegister

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

8.34 frame.cpp

[Go to the documentation of this file.](#)

```
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 extern volatile uint16_t eventRegister;
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 Frame frame_decode(const std::string& data) {
00023     try {
00024         Frame frame;
00025         std::stringstream ss(data);
00026         std::string token;
```

```

00068     std::getline(ss, token, DELIMITER);
00069     if (token != FRAME_BEGIN)
00070         throw std::runtime_error("Invalid frame header");
00071     frame.header = token;
00072
00073     std::string decoded_frame_data;
00074     while (std::getline(ss, token, DELIMITER)) {
00075         if (token == FRAME_END) break;
00076         decoded_frame_data += token + DELIMITER;
00077     }
00078     if (!decoded_frame_data.empty()) {
00079         decoded_frame_data.pop_back();
00080     }
00081
00082     std::stringstream frame_data_stream(decoded_frame_data);
00083
00084     std::getline(frame_data_stream, token, DELIMITER);
00085     frame.direction = std::stoi(token);
00086
00087     std::getline(frame_data_stream, token, DELIMITER);
00088     frame.operationType = string_to_operation_type(token);
00089
00090     std::getline(frame_data_stream, token, DELIMITER);
00091     frame.group = std::stoi(token);
00092
00093     std::getline(frame_data_stream, token, DELIMITER);
00094     frame.command = std::stoi(token);
00095
00096     std::getline(frame_data_stream, token, DELIMITER);
00097     frame.value = token;
00098
00099     std::getline(frame_data_stream, token, DELIMITER);
00100    frame.unit = token;
00101
00102    return frame;
00103 } catch (const std::exception& e) {
00104     uart_print("Frame error: " + std::string(e.what()), VerbosityLevel::ERROR);
00105     Frame error_frame = frame_build(OperationType::ERR, 0, 0, e.what());
00106     return error_frame;
00107 }
00108 }
00109
00110
00111 void frame_process(const std::string& data, Interface interface) {
00112     gpio_put(PICO_DEFAULT_LED_PIN, 0);
00113
00114     uart_print("Processing frame: " + data, VerbosityLevel::WARNING);
00115     try {
00116         Frame frame = frame_decode(data);
00117         uint32_t command_key = (static_cast<uint32_t>(frame.group) << 8) |
00118             static_cast<uint32_t>(frame.command);
00119
00120         std::vector<Frame> response_frames = execute_command(command_key, frame.value,
00121             frame.operationType);
00122
00123         gpio_put(PICO_DEFAULT_LED_PIN, 1);
00124
00125         // Send all responses through the same interface that received the command
00126         for (const auto& response_frame : response_frames) {
00127             if (interface == Interface::UART) {
00128                 send_frame_uart(response_frame);
00129             } else if (interface == Interface::LORA) {
00130                 send_frame_lora(response_frame);
00131                 sleep_ms(25);
00132             }
00133         }
00134     } catch (const std::exception& e) {
00135         Frame error_frame = frame_build(OperationType::ERR, 0, 0, e.what());
00136         if (interface == Interface::UART) {
00137             send_frame_uart(error_frame);
00138         } else if (interface == Interface::LORA) {
00139             send_frame_lora(error_frame);
00140         }
00141     }
00142 }
00143
00144
00145
00146 }
00147
00148 Frame frame_build(OperationType operation, uint8_t group, uint8_t command,
00149                      const std::string& value, const ValueUnit unitType) {
00150     Frame frame;
00151     frame.header = FRAME_BEGIN;
00152     frame.footer = FRAME_END;
00153
00154     switch (operation) {
00155         case OperationType::VAL:
00156             frame.direction = 1;
00157             frame.operationType = OperationType::VAL;
00158             frame.value = value;
00159
00160
00161
00162
00163
00164
00165
00166
00167
00168

```

```

00169     frame.unit = value_unit_type_to_string(unitType);
00170     break;
00171
00172     case OperationType::ERR:
00173         frame.direction = 1;
00174         frame.operationType = OperationType::ERR;
00175         frame.value = value;
00176         frame.unit = value_unit_type_to_string(ValueUnit::UNDEFINED);
00177         break;
00178
00179     case OperationType::RES:
00180         frame.direction = 1;
00181         frame.operationType = OperationType::RES;
00182         frame.value = value;
00183         frame.unit = value_unit_type_to_string(unitType);
00184         break;
00185
00186     case OperationType::SEQ:
00187         frame.direction = 1;
00188         frame.operationType = OperationType::SEQ;
00189         frame.value = value;
00190         frame.unit = value_unit_type_to_string(unitType);
00191         break;
00192     }
00193
00194     frame.group = group;
00195     frame.command = command;
00196
00197     return frame;
00198 }
00199 // end of FrameHandling group

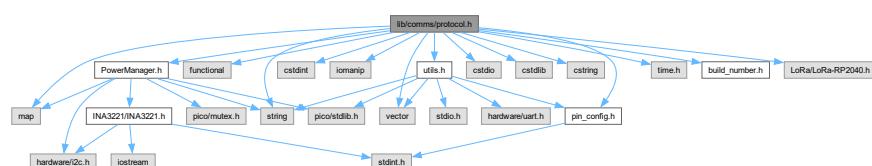
```

8.35 lib/comms/protocol.h File Reference

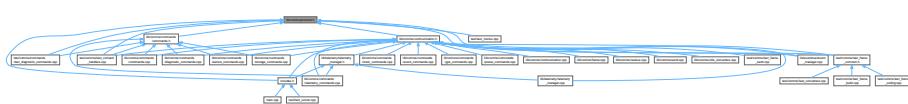
```

#include <string>
#include <map>
#include <functional>
#include <vector>
#include <cstdint>
#include <iomanip>
#include "pin_config.h"
#include "PowerManager.h"
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include "utils.h"
#include "time.h"
#include "build_number.h"
#include "LoRa/LoRa-RP2040.h"
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` {
 `PARAM_UNNECESSARY` , `PARAM_REQUIRED` , `PARAM_INVALID` , `INVALID_OPERATION` ,
 `NOT_ALLOWED` , `INVALID_FORMAT` , `INVALID_VALUE` , `FAIL_TO_SET` ,
 `INTERNAL_FAIL_TO_READ` , `UNKNOWN_ERROR` }

Standard error codes for command responses.

- enum class `OperationType` {
 `GET` , `SET` , `RES` , `VAL` ,
 `SEQ` , `ERR` }

Represents the type of operation being performed.

- enum class `CommandAccessLevel` { `NONE` , `READ_ONLY` , `WRITE_ONLY` , `READ_WRITE` }

Represents the access level required to execute a command.

- enum class `ValueUnit` {
 `UNDEFINED` , `SECOND` , `VOLT` , `BOOL` ,
 `DATETIME` , `TEXT` , `MILIAMP` }

Represents the unit of measurement for a payload value.

- enum class `ExceptionType` {
 `NONE` , `NOT_ALLOWED` , `INVALID_PARAM` , `INVALID_OPERATION` ,
 `PARAM_UNNECESSARY` }

Represents the type of exception that occurred during command execution.

- enum class `Interface` { `UART` , `LORA` }

Represents the communication interface being used.

Functions

- std::string `exception_type_to_string` (`ExceptionType` type)
Converts an `ExceptionType` to a string.
- std::string `error_code_to_string` (`ErrorCode` code)
Converts an `ErrorCode` to its string representation.
- std::string `operation_type_to_string` (`OperationType` type)
Converts an `OperationType` to a string.
- `OperationType string_to_operation_type` (const std::string &str)
Converts a string to an `OperationType`.
- std::vector< uint8_t > `hex_string_to_bytes` (const std::string &hexString)
Converts a hex string to a vector of bytes.
- std::string `value_unit_type_to_string` (`ValueUnit` unit)
Converts a `ValueUnit` to a string.

Variables

- const std::string `FRAME_BEGIN` = "KBST"
- const std::string `FRAME_END` = "TSBK"
- const char `DELIMITER` = ':';

8.35.1 Enumeration Type Documentation

8.35.1.1 ErrorCode

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

Standard error codes for command responses.

Enumerator

PARAM_UNNECESSARY	
PARAM_REQUIRED	
PARAM_INVALID	
INVALID_OPERATION	
NOT_ALLOWED	
INVALID_FORMAT	
INVALID_VALUE	
FAIL_TO_SET	
INTERNAL_FAIL_TO_READ	
UNKNOWN_ERROR	

Definition at line 45 of file [protocol.h](#).

8.35.1.2 OperationType

```
enum class OperationType [strong]
```

Represents the type of operation being performed.

Enumerator

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

Definition at line 63 of file [protocol.h](#).

8.35.1.3 CommandAccessLevel

```
enum class CommandAccessLevel [strong]
```

Represents the access level required to execute a command.

Enumerator

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

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

8.35.1.4 ValueUnit

```
enum class ValueUnit [strong]
```

Represents the unit of measurement for a payload value.

Enumerator

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

Definition at line 102 of file [protocol.h](#).

8.35.1.5 ExceptionType

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

Represents the type of exception that occurred during command execution.

Enumerator

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

Definition at line 125 of file [protocol.h](#).

8.35.1.6 Interface

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

Represents the communication interface being used.

Enumerator

UART	UART interface.
LORA	LoRa interface.

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

8.35.2 Function Documentation**8.35.2.1 exception_type_to_string()**

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

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

Parameters

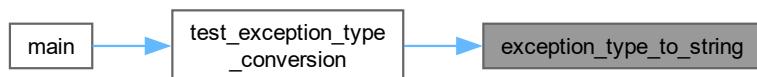
<i>type</i>	The ExceptionType to convert.
-------------	---

Returns

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

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

Here is the caller graph for this function:



8.35.2.2 error_code_to_string()

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

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

Parameters

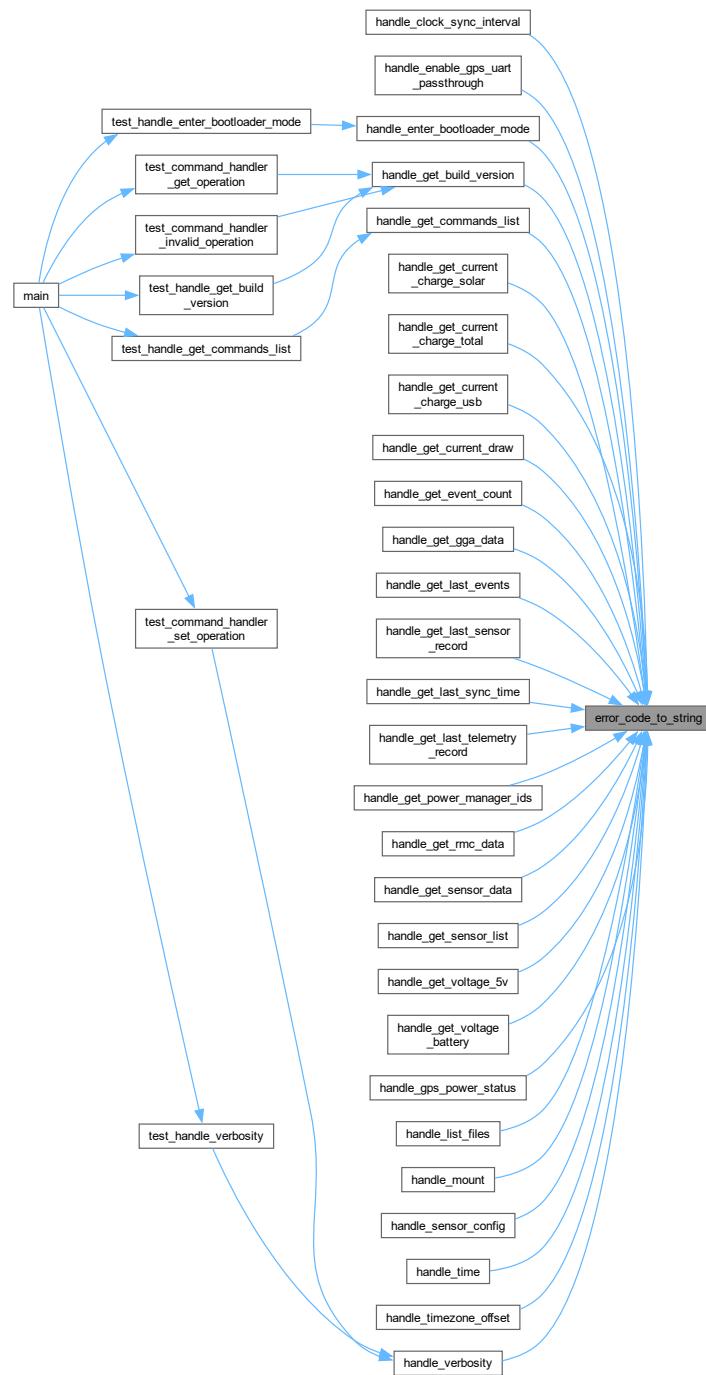
<i>code</i>	The error code
-------------	----------------

Returns

String representation of the error code

Definition at line 83 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



8.35.2.3 `operation_type_to_string()`

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

Converts an `OperationType` to a string.

Parameters

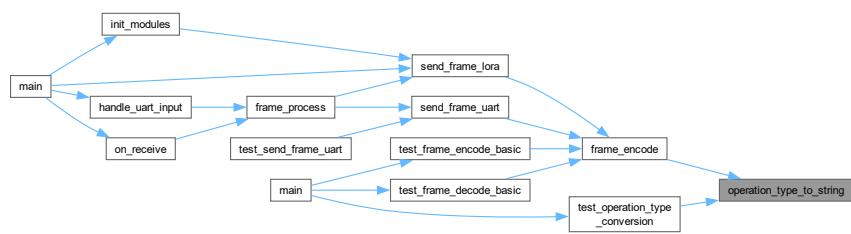
<code>type</code>	The <code>OperationType</code> to convert.
-------------------	--

Returns

The string representation of the `OperationType`.

Definition at line 50 of file `utils_converters.cpp`.

Here is the caller graph for this function:

**8.35.2.4 string_to_operation_type()**

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

Converts a string to an `OperationType`.

Parameters

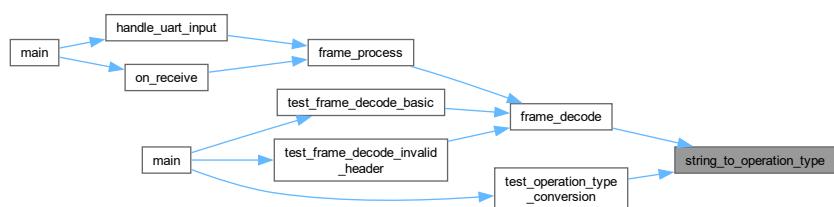
<code>str</code>	The string to convert.
------------------	------------------------

Returns

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

Definition at line 68 of file `utils_converters.cpp`.

Here is the caller graph for this function:



8.35.2.5 hex_string_to_bytes()

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

Converts a hex string to a vector of bytes.

Parameters

<i>hexString</i>	The hex string to convert.
------------------	----------------------------

Returns

A vector of bytes representing the hex string.

Definition at line 104 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



8.35.2.6 value_unit_type_to_string()

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

Converts a [ValueUnit](#) to a string.

Parameters

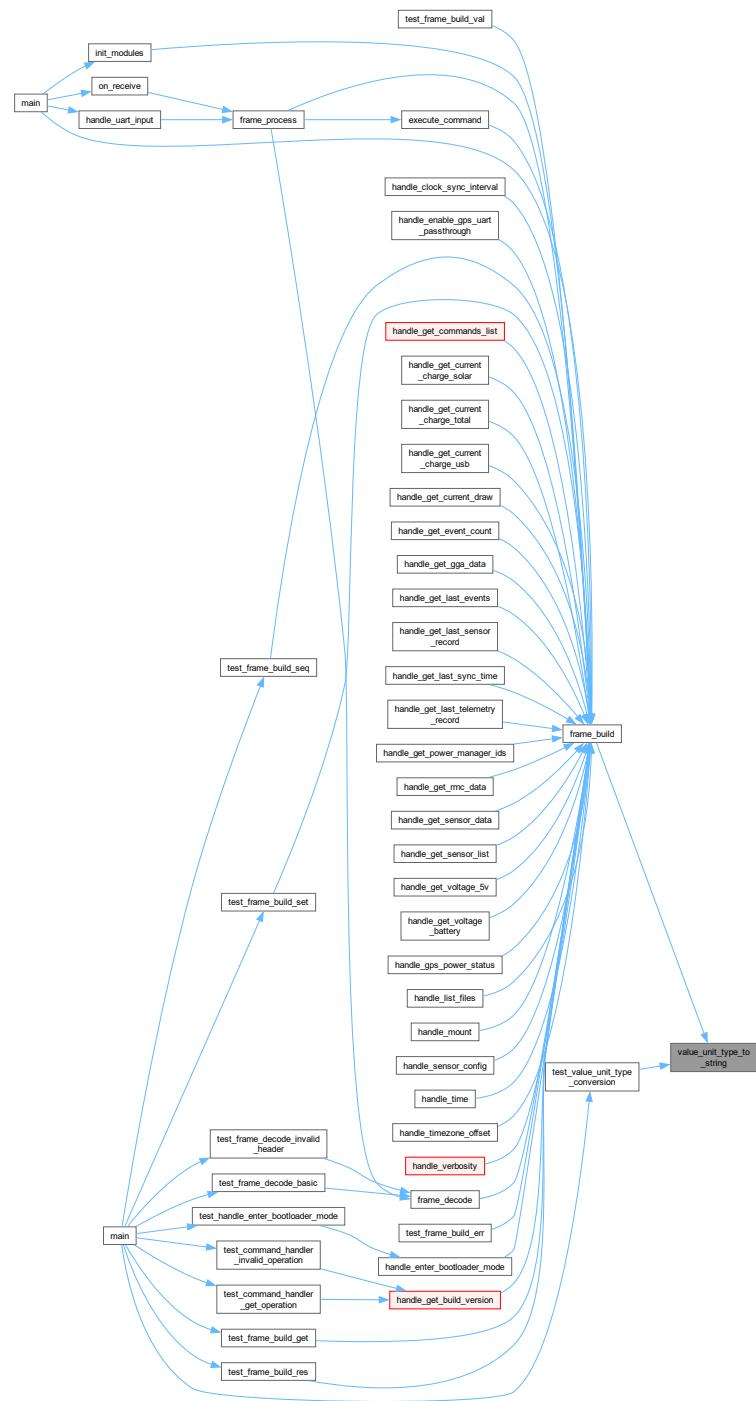
<i>unit</i>	The ValueUnit to convert.
-------------	---

Returns

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

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

Here is the caller graph for this function:



8.35.3 Variable Documentation

8.35.3.1 FRAME_BEGIN

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

Definition at line 26 of file [protocol.h](#).

8.35.3.2 FRAME_END

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

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

8.35.3.3 DELIMITER

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

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

8.36 protocol.h

[Go to the documentation of this file.](#)

```
00001 // protocol.h
00002 #ifndef PROTOCOL_H
00003 #define PROTOCOL_H
00004
00005 #include <string>
00006 #include <map>
00007 #include <functional>
00008 #include <vector>
00009 #include <cstdint>
00010 #include <iomanip>
00011 #include "pin_config.h"
00012 #include "PowerManager.h"
00013 #include <cstdio>
00014 #include <cstdlib>
00015 #include <map>
00016 #include <cstring>
00017 #include "utils.h"
00018 #include "time.h"
00019 #include "build_number.h"
00020 #include "LoRa/LoRa-RP2040.h"
00021
00026 const std::string FRAME_BEGIN = "KBST";
00027
00032 const std::string FRAME_END = "TSBK";
00033
00038 const char DELIMITER = ';';
00039
00040
00045 enum class ErrorCode {
00046     PARAM_UNNECESSARY,           // Parameter provided but not needed
00047     PARAM_REQUIRED,              // Required parameter missing
00048     PARAM_INVALID,               // Parameter has invalid format or value
00049     INVALID_OPERATION,           // Operation not allowed for this command
00050     NOT_ALLOWED,                 // Operation not permitted
00051     INVALID_FORMAT,              // Input format is incorrect
00052     INVALID_VALUE,                // Value is outside expected range
00053     FAIL_TO_SET,                  // Failed to set provided value
00054     INTERNAL_FAIL_TO_READ,        // Failed to read from device in remote
00055     UNKNOWN_ERROR                 // Generic error
00056 };
00057
00058
```

```

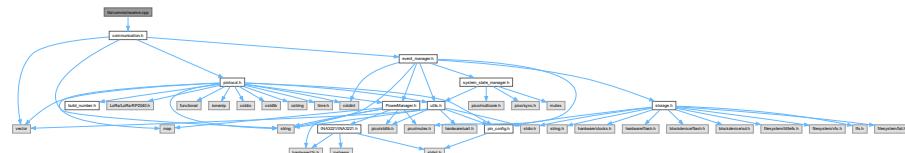
00063 enum class OperationType {
00065     GET,
00067     SET,
00069     RES,
00071     VAL,
00073     SEQ,
00075     ERR,
00076 };
00078
00079
00080
00085 enum class CommandAccessLevel {
00087     NONE,
00089     READ_ONLY,
00091     WRITE_ONLY,
00093     READ_WRITE
00094 };
00095
00096
00097
00102 enum class ValueUnit {
00104     UNDEFINED,
00106     SECOND,
00108     VOLT,
00110     BOOL,
00112     DATETIME,
00114     TEXT,
00116     MILIAMP,
00117 };
00118
00119
00120
00125 enum class ExceptionType {
00127     NONE,
00129     NOT_ALLOWED,
00131     INVALID_PARAM,
00133     INVALID_OPERATION,
00135     PARAM_UNNECESSARY
00136 };
00137
00138
00139
00144 enum class Interface {
00146     UART,
00148     LORA
00149 };
00150
00151
00227 struct Frame {
00228     std::string header;           // Start marker
00229     uint8_t direction;          // 0 = ground->sat, 1 = sat->ground
00230     OperationType operationType; // Operation type
00231     uint8_t group;              // Group ID
00232     uint8_t command;            // Command ID within group
00233     std::string value;          // Payload value
00234     std::string unit;           // Payload unit
00235     std::string footer;         // End marker
00236 };
00237
00238 std::string exception_type_to_string(ExceptionType type);
00239 std::string error_code_to_string(ErrorCode code);
00240 std::string operation_type_to_string(OperationType type);
00241 OperationType string_to_operation_type(const std::string& str);
00242 std::vector<uint8_t> hex_string_to_bytes(const std::string& hexString);
00243 std::string value_unit_type_to_string(ValueUnit unit);
00244
00245 #endif

```

8.37 lib/comms/receive.cpp File Reference

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

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



Macros

- `#define MAX_PACKET_SIZE 255`

Functions

- `void on_receive (int packet_size)`
Callback function for handling received LoRa packets.
- `void handle_uart_input ()`
Handles UART input.

8.37.1 Detailed Description

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

Definition in file [receive.cpp](#).

8.37.2 Macro Definition Documentation

8.37.2.1 MAX_PACKET_SIZE

```
#define MAX_PACKET_SIZE 255
```

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

8.37.3 Function Documentation

8.37.3.1 on_receive()

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

Callback function for handling received LoRa packets.

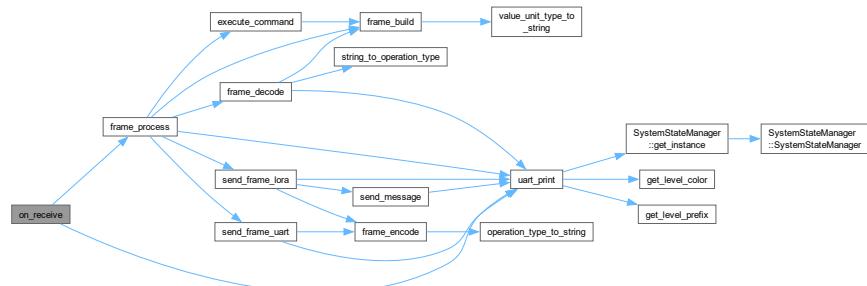
Parameters

<code>packet_size</code>	The size of the received packet.
--------------------------	----------------------------------

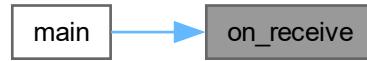
Reads the received LoRa packet, extracts metadata, validates the lora_address_remote and local addresses, extracts the frame data, and processes it. Prints raw hex values for debugging.

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

Here is the call graph for this function:



Here is the caller graph for this function:



8.37.3.2 handle_uart_input()

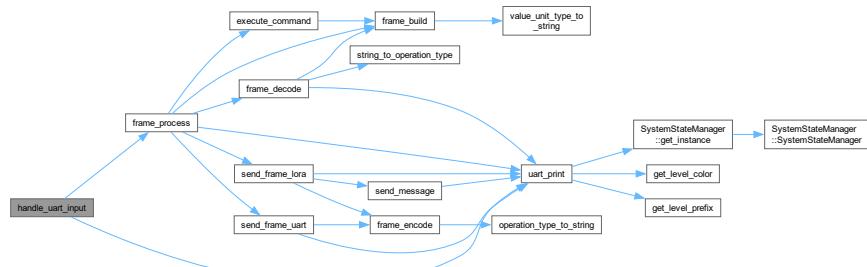
```
void handle_uart_input ()
```

Handles UART input.

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

Definition at line 86 of file [receive.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.38 receive.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003 #define MAX_PACKET_SIZE 255
00004
00009
00016 void on_receive(int packet_size) {
00017     if (packet_size == 0) return;
00018     uart_print("Received LoRa packet of size " + std::to_string(packet_size), VerbosityLevel::DEBUG);
00019
00020     std::vector<uint8_t> buffer;
00021     buffer.reserve(packet_size);
00022
00023     int bytes_read = 0;
00024
00025     while (LoRa.available() && bytes_read < packet_size) {
00026         if (bytes_read >= MAX_PACKET_SIZE) {
00027             uart_print("Error: Packet exceeds maximum allowed size!", VerbosityLevel::ERROR);
00028             return;
00029         }
00030         buffer.push_back(LoRa.read());
00031         bytes_read++;
00032     }
00033
00034     if (bytes_read < 2) {
00035         uart_print("Error: Packet too small to contain metadata!", VerbosityLevel::ERROR);
00036         return;
00037     }
00038
00039     uart_print("Received " + std::to_string(bytes_read) + " bytes", VerbosityLevel::DEBUG);
00040
00041     uint8_t received_destination = buffer[0];
00042     uint8_t received_local_address = buffer[1];
00043
00044     if (received_destination != lora_address_local) {
00045         uart_print("Error: Destination address mismatch!", VerbosityLevel::ERROR);
00046         return;
00047     }
00048
00049     if (received_local_address != lora_address_remote) {
00050         uart_print("Error: Local address mismatch!", VerbosityLevel::ERROR);
00051         return;
00052     }
00053
00054     int start_index = 2;
00055
00056     std::string received = std::string(reinterpret_cast<char*>(buffer + start_index), bytes_read -
00057                                         start_index);
00058
00059     if (received.empty()) return;
00060
00061     std::stringstream hex_dump;
00062     hex_dump << "Raw bytes: ";
00063     for (int i = 0; i < bytes_read; i++) {
00064         hex_dump << std::hex << std::setfill('0') << std::setw(2)
00065             << static_cast<int>(buffer[i]) << " ";
00066     }
00067     uart_print(hex_dump.str(), VerbosityLevel::DEBUG);
00068
00069     size_t header_pos = received.find(FRAME_BEGIN);
00070     size_t footer_pos = received.find(FRAME_END);

```

```

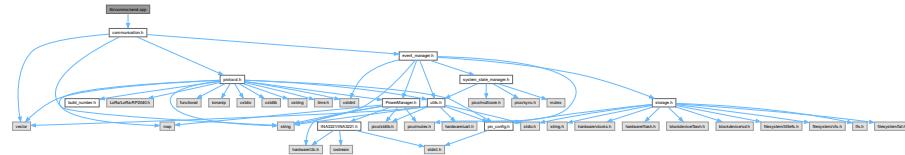
00070     if (header_pos != std::string::npos && footer_pos != std::string::npos && footer_pos > header_pos)
00071     {
00072         std::string frame_data = received.substr(header_pos, footer_pos + FRAME_END.length() -
00073             header_pos);
00073         uart_print("Extracted frame (length=" + std::to_string(frame_data.length()) + "): " +
00074             frame_data, VerbosityLevel::DEBUG);
00075         frame_process(frame_data, Interface::LORA);
00076     } else {
00077         uart_print("No valid frame found in received data", VerbosityLevel::WARNING);
00078     }
00079
00080
00086 void handle_uart_input() {
00087     static std::string uart_buffer;
00088
00089     while (uart_is_readable(DEBUG_UART_PORT)) {
00090         char c = uart_getc(DEBUG_UART_PORT);
00091
00092         if (c == '\r' || c == '\n') {
00093             uart_print("Received UART string: " + uart_buffer, VerbosityLevel::DEBUG);
00094             frame_process(uart_buffer, Interface::UART);
00095             uart_buffer.clear();
00096         } else {
00097             uart_buffer += c;
00098         }
00099     }
00100 }
```

8.39 lib/comms/send.cpp File Reference

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

#include "communication.h"

Include dependency graph for send.cpp:



Functions

- void [send_message](#) (string *outgoing*)
Sends a message using LoRa.
- void [send_frame_lora](#) (const Frame &*frame*)
- void [send_frame_uart](#) (const Frame &*frame*)

8.39.1 Detailed Description

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

Definition in file [send.cpp](#).

8.39.2 Function Documentation

8.39.2.1 send_message()

```
void send_message (
    string outgoing)
```

Sends a message using LoRa.

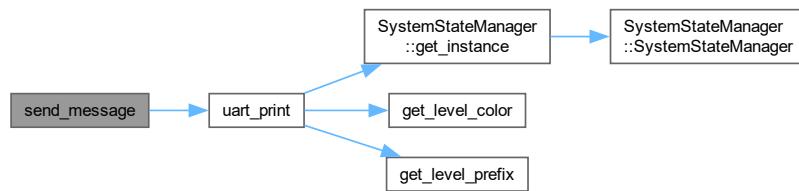
Parameters

<i>outgoing</i>	The message to send.
-----------------	----------------------

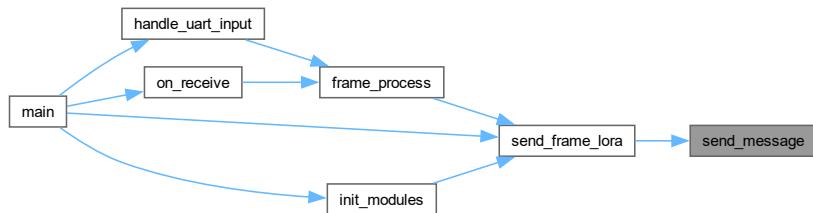
Converts the outgoing string to a C-style string, adds destination and local addresses, and sends the message using LoRa. Prints a log message to the UART.

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

Here is the call graph for this function:



Here is the caller graph for this function:

**8.39.2.2 send_frame_lora()**

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

Definition at line 39 of file [send.cpp](#).

8.39.2.3 send_frame_uart()

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

Definition at line 46 of file [send.cpp](#).

8.40 send.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003
00008
00015 void send_message(string outgoing)
00016 {
00017     std::vector<char> send(outgoing.length() + 1);
00018     strcpy(send.data(), outgoing.c_str());
00019
00020     uart_print("LoRa packet begin", VerbosityLevel::DEBUG);
00021     LoRa.beginPacket(); // start packet
00022     LoRa.write(lora_address_remote); // add destination address
00023     LoRa.write(lora_address_local); // add sender address
00024     LoRa.print(send.data()); // add payload
00025     LoRa.endPacket(false); // finish packet and send it
00026
00027     uart_print("LoRa packet end", VerbosityLevel::DEBUG);
00028
00029     std::string message_to_log = "Sent message of size " + std::to_string(send.size());
00030     message_to_log += " to 0x" + std::to_string(lora_address_remote);
00031     message_to_log += " containing: " + string(send.data());
00032
00033     uart_print(message_to_log, VerbosityLevel::DEBUG);
00034
00035     LoRa.flush();
00036 }
00037
00038
00039 void send_frame_lora(const Frame& frame) {
00040     uart_print("Sending frame via LoRa", VerbosityLevel::DEBUG);
00041     string outgoing = frame_encode(frame);
00042     send_message(outgoing);
00043     uart_print("Frame sent via LoRa", VerbosityLevel::DEBUG);
00044 }
00045
00046 void send_frame_uart(const Frame& frame) {
00047     std::string encoded_frame = frame_encode(frame);
00048     uart_print(encoded_frame);
00049 }
00050
00051

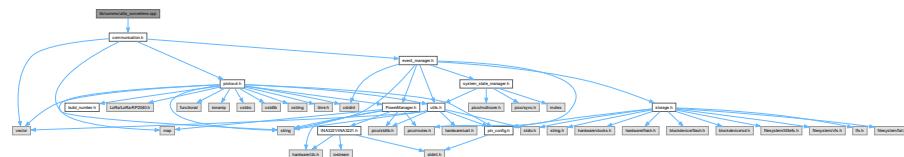
```

8.41 lib/comms/utils_converters.cpp File Reference

Implements utility functions for converting between different data types.

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

Include dependency graph for utils_converters.cpp:



Functions

- `std::string exception_type_to_string (ExceptionType type)`
Converts an `ExceptionType` to a string.
- `std::string value_unit_type_to_string (ValueUnit unit)`
Converts a `ValueUnit` to a string.
- `std::string operation_type_to_string (OperationType type)`

- Converts an `OperationType` to a `string`.
- `OperationType string_to_operation_type (const std::string &str)`
Converts a `string` to an `OperationType`.
- `std::string error_code_to_string (ErrorCode code)`
Converts an `ErrorCode` to its `string` representation.
- `std::vector< uint8_t > hex_string_to_bytes (const std::string &hexString)`
Converts a `hex string` to a `vector of bytes`.

8.41.1 Detailed Description

Implements utility functions for converting between different data types.

Definition in file `utils_converters.cpp`.

8.41.2 Function Documentation

8.41.2.1 exception_type_to_string()

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

Converts an `ExceptionType` to a `string`.

Parameters

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

Returns

The `string` representation of the `ExceptionType`.

Definition at line 14 of file `utils_converters.cpp`.

Here is the caller graph for this function:



8.41.2.2 value_unit_type_to_string()

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

Converts a `ValueUnit` to a `string`.

Parameters

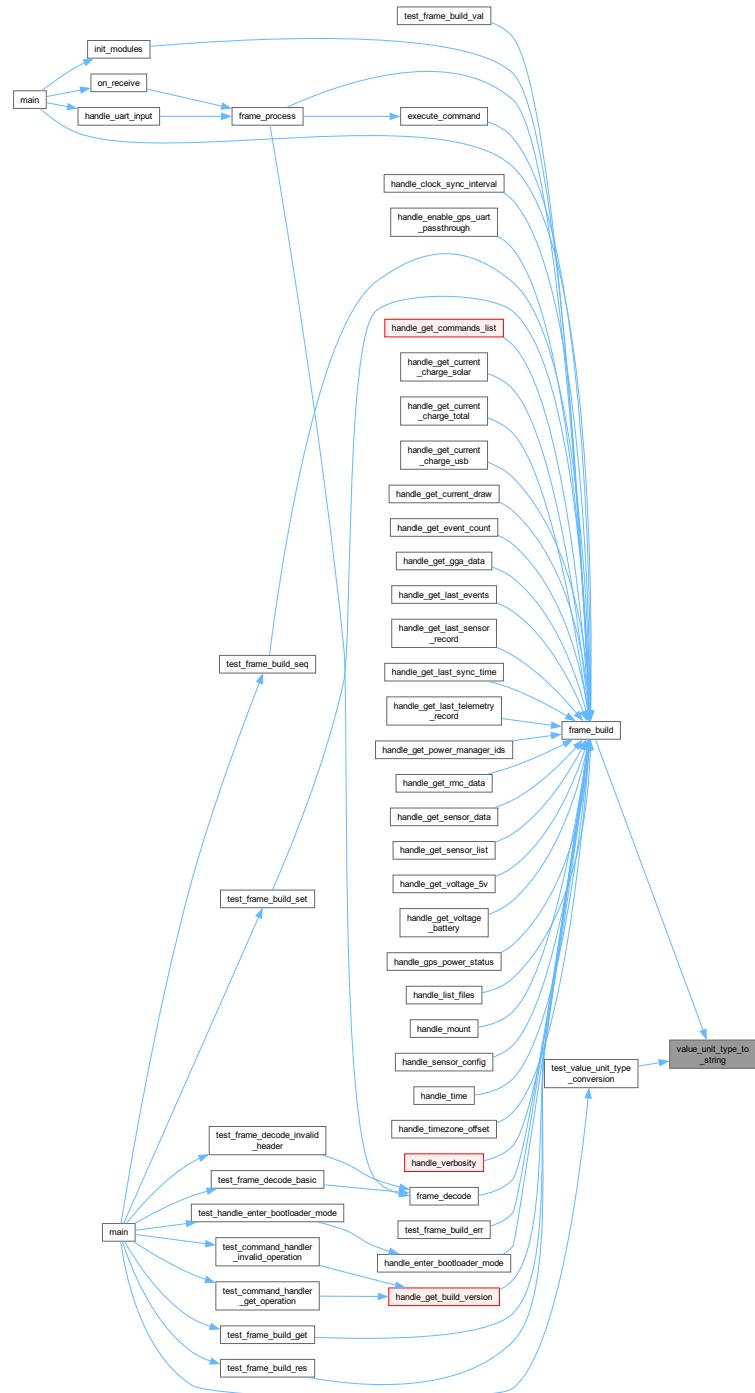
`unit` | The [ValueUnit](#) to convert.

Returns

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

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

Here is the caller graph for this function:



8.41.2.3 operation_type_to_string()

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

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

Parameters

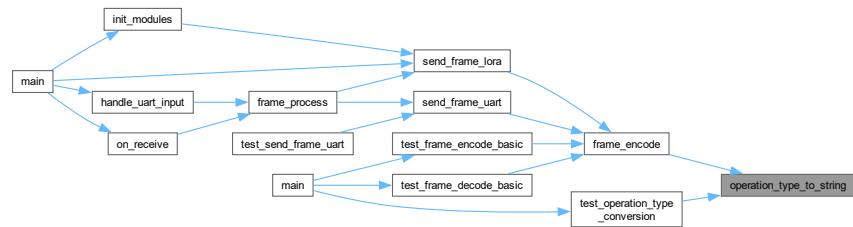
<i>type</i>	The OperationType to convert.
-------------	---

Returns

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

Definition at line 50 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



8.41.2.4 string_to_operation_type()

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

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

Parameters

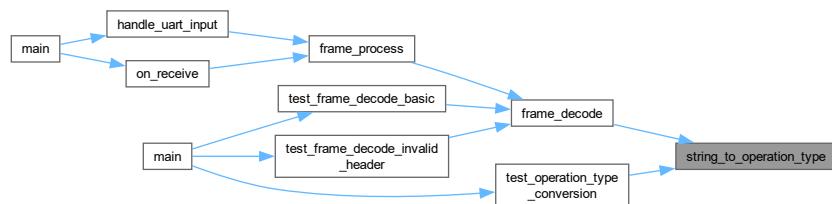
<i>str</i>	The string to convert.
------------	------------------------

Returns

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

Definition at line 68 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



8.41.2.5 error_code_to_string()

```
std::string error_code_to_string (
```

<code>ErrorCode</code>	<code>code</code>
------------------------	-------------------

```
)
```

Converts an `ErrorCode` to its string representation.

Parameters

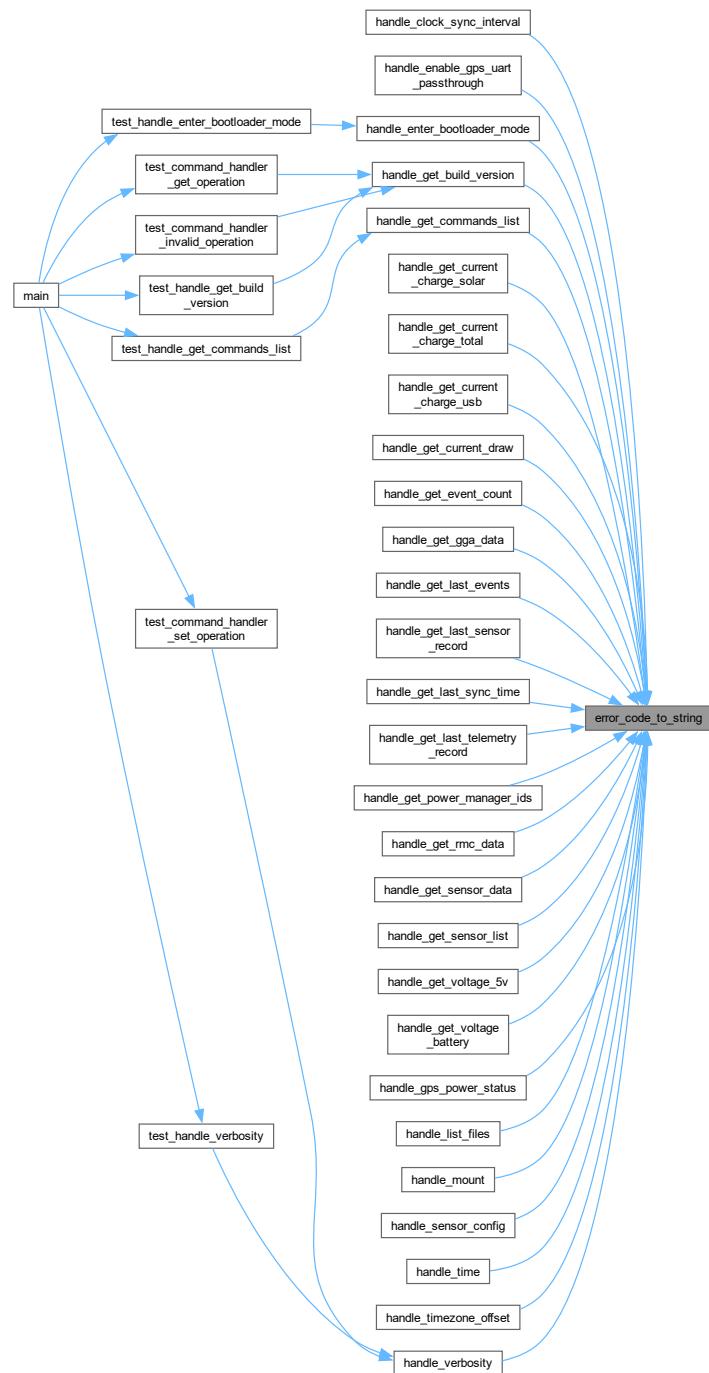
<code>code</code>	The error code
-------------------	----------------

Returns

String representation of the error code

Definition at line [83](#) of file [utils_converters.cpp](#).

Here is the caller graph for this function:



8.41.2.6 hex_string_to_bytes()

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

Converts a hex string to a vector of bytes.

Parameters

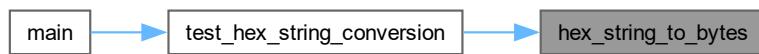
<i>hexString</i>	The hex string to convert.
------------------	----------------------------

Returns

A vector of bytes representing the hex string.

Definition at line 104 of file [utils_converters.cpp](#).

Here is the caller graph for this function:



8.42 utils_converters.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003
00008
00014 std::string exception_type_to_string(ExceptionType type) {
00015     switch (type) {
00016         case ExceptionType::NOT_ALLOWED:      return "NOT ALLOWED";
00017         case ExceptionType::INVALID_PARAM:    return "INVALID PARAM";
00018         case ExceptionType::INVALID_OPERATION: return "INVALID OPERATION";
00019         case ExceptionType::PARAM_UNNECESSARY: return "PARAM UNECESSARY";
00020         case ExceptionType::NONE:             return "NONE";
00021         default:                            return "UNKNOWN EXCEPTION";
00022     }
00023 }
00024
00025
00031 std::string value_unit_type_to_string(ValueUnit unit) {
00032     switch (unit) {
00033         case ValueUnit::UNDEFINED:   return "";
00034         case ValueUnit::SECOND:     return "s";
00035         case ValueUnit::VOLT:       return "V";
00036         case ValueUnit::BOOL:       return "";
00037         case ValueUnit::DATETIME:   return "";
00038         case ValueUnit::TEXT:       return "";
00039         case ValueUnit::MILIAMP:    return "mA";
00040         default:                  return "";
00041     }
00042 }
00043
00044
00050 std::string operation_type_to_string(OperationType type) {
00051     switch (type) {
00052         case OperationType::GET:   return "GET";
00053         case OperationType::SET:   return "SET";
00054         case OperationType::VAL:   return "VAL";
00055         case OperationType::ERR:   return "ERR";
00056         case OperationType::RES:   return "RES";
00057         case OperationType::SEQ:   return "SEQ";
00058         default:                 return "UNKNOWN";
00059     }
00060 }
00061
00062
00068 OperationType string_to_operation_type(const std::string& str) {
00069     if (str == "GET") return OperationType::GET;

```

```

00070     if (str == "SET") return OperationType::SET;
00071     if (str == "VAL") return OperationType::VAL;
00072     if (str == "ERR") return OperationType::ERR;
00073     if (str == "RES") return OperationType::RES;
00074     if (str == "SEQ") return OperationType::SEQ;
00075     return OperationType::GET; // Default to GET
00076 }
00077
00083 std::string error_code_to_string(ErrorCode code) {
00084     switch (code) {
00085         case ErrorCode::PARAM_UNNECESSARY:      return "PARAM_UNNECESSARY";
00086         case ErrorCode::PARAM_REQUIRED:        return "PARAM_REQUIRED";
00087         case ErrorCode::PARAM_INVALID:         return "PARAM_INVALID";
00088         case ErrorCode::INVALID_OPERATION:    return "INVALID_OPERATION";
00089         case ErrorCode::NOT_ALLOWED:          return "NOT_ALLOWED";
00090         case ErrorCode::INVALID_FORMAT:       return "INVALID_FORMAT";
00091         case ErrorCode::INVALID_VALUE:        return "INVALID_VALUE";
00092         case ErrorCode::FAIL_TO_SET:          return "FAIL_TO_SET";
00093         case ErrorCode::INTERNAL_FAIL_TO_READ: return "INTERNAL_FAIL_TO_READ";
00094     default:                                return "UNKNOWN_ERROR";
00095 }
00096 }
00097
00098
00104 std::vector<uint8_t> hex_string_to_bytes(const std::string& hexString) {
00105     std::vector<uint8_t> bytes;
00106     for (size_t i = 0; i < hexString.length(); i += 2) {
00107         std::string byteString = hexString.substr(i, 2);
00108         unsigned int byte;
00109         std::stringstream ss;
00110         ss << std::hex << byteString;
00111         ss >> byte;
00112         bytes.push_back(static_cast<uint8_t>(byte));
00113     }
00114     return bytes;
00115 }

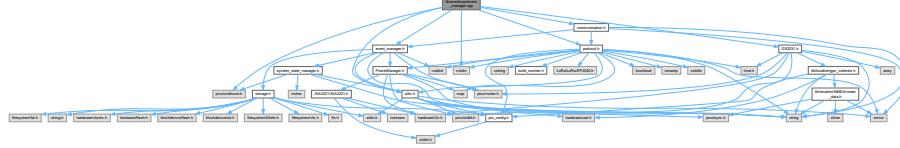
```

8.43 lib/eventman/event_manager.cpp File Reference

Implements the event management system for the Kubisat firmware.

```
#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:



Variables

- volatile uint16_t **eventLogId** = 0
Global event log ID counter.
- DS3231 **systemClock**
External declaration of the system clock.
- EventManagerImpl **eventManager**
Global instance of the `EventManager` implementation.

8.43.1 Detailed Description

Implements the event management system for the Kabisat firmware.

This file contains the implementation for logging events, managing event storage, and checking for specific events such as power status changes.

Definition in file [event_manager.cpp](#).

8.44 event_manager.cpp

[Go to the documentation of this file.](#)

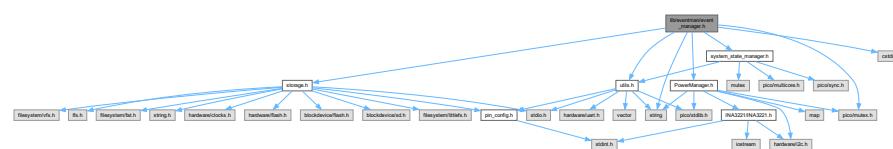
```
00001 #include "event_manager.h"
00002 #include <cstdio>
00003 #include "protocol.h"
00004 #include "pico/multicore.h"
00005 #include "communication.h"
00006 #include "utils.h"
00007 #include "DS3231.h"
00008
00016
00021 volatile uint16_t eventLogId = 0;
00022
00027 extern DS3231 systemClock;
00028
00033 EventManagerImpl eventManager;
00034
00035 uint16_t EventManager::nextEventId = 0;
00036
00045 void EventManager::log_event(uint8_t group, uint8_t event) {
00046     mutex_enter_blocking(&eventMutex);
00047
00048     // Clear buffer if it's full
00049     if (eventCount >= EVENT_BUFFER_SIZE) {
00050         eventCount = 0;
00051         writeIndex = 0;
00052     }
00053
00054     EventLog& log = events[writeIndex];
00055     log.id = nextEventId++;
00056     log.timestamp = systemClock.get_unix_time();
00057     log.group = group;
00058     log.event = event;
00059
00060     // Print event immediately
00061     uart_print(log.to_string(), VerbosityLevel::WARNING);
00062
00063     writeIndex = (writeIndex + 1) % EVENT_BUFFER_SIZE;
00064     eventCount++;
00065     eventsSinceFlush++;
00066
00067     // Flush to storage every EVENT_FLUSH_THRESHOLD events
00068     if (eventsSinceFlush >= EVENT_FLUSH_THRESHOLD) {
00069         save_to_storage();
00070         eventsSinceFlush = 0;
00071     }
00072
00073     mutex_exit(&eventMutex);
00074 }
00075
00076
00083 const EventLog& EventManager::get_event(size_t index) const {
00084     static const EventLog emptyEvent = {0, 0, 0, 0}; // Initialize {id, timestamp, group, event}
00085     if (index >= eventCount) {
00086         return emptyEvent;
00087     }
00088
00089     // Calculate actual index in circular buffer
00090     size_t actualIndex;
00091     if (eventCount == EVENT_BUFFER_SIZE) {
00092         actualIndex = (writeIndex + index) % EVENT_BUFFER_SIZE;
00093     } else {
00094         actualIndex = index;
00095     }
00096
00097     return events[actualIndex];
00098 }
```

8.45 lib/eventman/event_manager.h File Reference

Manages the event logging system for the Kabisat firmware.

```
#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](#)
Represents a single event log entry.
- class [EventManager](#)
Manages the event logging system.
- class [EventManagerImpl](#)
Implementation of the [EventManager](#) class.
- class [EventEmitter](#)
Provides a static method for emitting events.

Macros

- #define [EVENT_BUFFER_SIZE](#) 100
- #define [EVENT_FLUSH_THRESHOLD](#) 10
- #define [EVENT_LOG_FILE](#) "/event_log.csv"

Enumerations

- enum class `EventGroup` : `uint8_t` {

`EventGroup::SYSTEM = 0x00 , EventGroup::POWER = 0x01 , EventGroup::COMMS = 0x02 ,`

`EventGroup::GPS = 0x03 ,`

`EventGroup::CLOCK = 0x04 }`

Represents the group to which an event belongs.
- enum class `SystemEvent` : `uint8_t` {

`SystemEvent::BOOT = 0x01 , SystemEvent::SHUTDOWN = 0x02 , SystemEvent::WATCHDOG_RESET =`

`0x03 , SystemEvent::CORE1_START = 0x04 ,`

`SystemEvent::CORE1_STOP = 0x05 }`

Represents specific system events.
- enum class `PowerEvent` : `uint8_t` {

`PowerEvent::LOW_BATTERY = 0x01 , PowerEvent::OVERCHARGE = 0x02 , PowerEvent::POWER_FALLING`

`= 0x03 , PowerEvent::POWER_NORMAL = 0x04 ,`

`PowerEvent::SOLAR_ACTIVE = 0x05 , PowerEvent::SOLAR_INACTIVE = 0x06 , PowerEvent::USB_CONNECTED`

`= 0x07 , PowerEvent::USB_DISCONNECTED = 0x08 }`

Represents specific power-related 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 }`

Represents specific communication-related 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 }`

Represents specific GPS-related events.
- enum class `ClockEvent` : `uint8_t` { `ClockEvent::CHANGED = 0x01 , ClockEvent::GPS_SYNC = 0x02 ,`

`ClockEvent::GPS_SYNC_DATA_NOT_READY = 0x03 }`

Represents specific clock-related events.

Functions

- class `EventLog __attribute__ ((packed))`
- `std::string to_string () const`

Converts the `EventLog` to a string representation.

Variables

- `uint16_t id`

Sequence number.
- `uint32_t timestamp`

Unix timestamp or system time.
- `uint8_t group`

Event group identifier.
- `uint8_t event`

Specific event identifier.
- class `EventManager __attribute__`
- `EventManagerImpl eventManager`

Global instance of the `EventManagerImpl` class.

8.45.1 Detailed Description

Manages the event logging system for the Kabisat firmware.

Definition in file [event_manager.h](#).

8.45.2 Macro Definition Documentation

8.45.2.1 EVENT_BUFFER_SIZE

```
#define EVENT_BUFFER_SIZE 100
```

Definition at line [13](#) of file [event_manager.h](#).

8.45.2.2 EVENT_FLUSH_THRESHOLD

```
#define EVENT_FLUSH_THRESHOLD 10
```

Definition at line [14](#) of file [event_manager.h](#).

8.45.2.3 EVENT_LOG_FILE

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

Definition at line [15](#) of file [event_manager.h](#).

8.45.3 Function Documentation

8.45.3.1 to_string()

```
std::string __attribute__::__to_string () const
```

Converts the [EventLog](#) to a string representation.

Returns

A string representation of the [EventLog](#).

Definition at line [14](#) of file [event_manager.h](#).

8.45.4 Variable Documentation

8.45.4.1 id

```
uint16_t id
```

Sequence number.

Definition at line [2](#) of file [event_manager.h](#).

8.45.4.2 timestamp

```
uint32_t timestamp
```

Unix timestamp or system time.

Definition at line 4 of file [event_manager.h](#).

8.45.4.3 group

```
uint8_t group
```

Event group identifier.

Definition at line 6 of file [event_manager.h](#).

8.45.4.4 event

```
uint8_t event
```

Specific event identifier.

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

8.46 event_manager.h

[Go to the documentation of this file.](#)

```
00001 #ifndef EVENT_MANAGER_H
00002 #define EVENT_MANAGER_H
00003
00004 #include "PowerManager.h"
00005 #include <cstdint>
00006 #include <string>
00007 #include "pico/mutex.h"
00008 #include "storage.h"
00009 #include "utils.h"
00010 #include "system_state_manager.h"
00011
00012
00013 #define EVENT_BUFFER_SIZE 100
00014 #define EVENT_FLUSH_THRESHOLD 10
00015 #define EVENT_LOG_FILE "/event_log.csv"
00016
00024
00025
00030 enum class EventGroup : uint8_t {
00032     SYSTEM = 0x00,
00034     POWER = 0x01,
00036     COMMS = 0x02,
00038     GPS = 0x03,
00040     CLOCK = 0x04
00041 };
00042
00047 enum class SystemEvent : uint8_t {
00049     BOOT = 0x01,
00051     SHUTDOWN = 0x02,
00053     WATCHDOG_RESET = 0x03,
00055     CORE1_START = 0x04,
00057     CORE1_STOP = 0x05
00058 };
00059
00064 enum class PowerEvent : uint8_t {
00066     LOW_BATTERY = 0x01,
00068     OVERCHARGE = 0x02,
```

```

00070     POWER_FALLING      = 0x03,
00072     POWER_NORMAL       = 0x04,
00074     SOLAR_ACTIVE        = 0x05,
00076     SOLAR_INACTIVE      = 0x06,
00078     USB_CONNECTED       = 0x07,
00080     USB_DISCONNECTED    = 0x08
00081 };
00082
00087 enum class CommsEvent : uint8_t {
00089     RADIO_INIT          = 0x01,
00091     RADIO_ERROR         = 0x02,
00093     MSG_RECEIVED        = 0x03,
00095     MSG_SENT            = 0x04,
00097     UART_ERROR          = 0x06
00098 };
00099
00104 enum class GPSEvent : uint8_t {
00106     LOCK                = 0x01,
00108     LOST                = 0x02,
00110     ERROR               = 0x03,
00112     POWER_ON             = 0x04,
00114     POWER_OFF            = 0x05,
00116     DATA_READY           = 0x06,
00118     PASS_THROUGH_START   = 0x07,
00120     PASS_THROUGH_END     = 0x08
00121 };
00122
00123
00128 enum class ClockEvent : uint8_t {
00130     CHANGED             = 0x01,
00132     GPS_SYNC             = 0x02,
00134     GPS_SYNC_DATA_NOT_READY = 0x03
00135 };
00136
00137
00142 class EventLog {
00143     public:
00145         uint16_t id;
00147         uint32_t timestamp;
00149         uint8_t group;
00151         uint8_t event;
00152
00157         std::string to_string() const {
00158             char buffer[256] = {0};
00159             snprintf(buffer, sizeof(buffer),
00160                     "EventLog: id=%u, timestamp=%lu, group=%u, event=%u",
00161                     id, timestamp, group, event);
00162             return std::string(buffer);
00163         }
00164     } __attribute__((packed));
00165
00166
00171 class EventManager {
00172     public:
00173         EventManager()
00174             : eventCount(0)
00175             , writeIndex(0)
00176             , eventsSinceFlush(0) // Add eventsSinceFlush initialization
00177         {
00178             mutex_init(&eventMutex);
00179         }
00180
00181         virtual ~EventManager() = default;
00182
00183         virtual void init() {
00184             load_from_storage();
00185         }
00186
00187         void log_event(uint8_t group, uint8_t event);
00188
00189         const EventLog& get_event(size_t index) const;
00190
00191         size_t get_event_count() const { return eventCount; }
00192
00193         virtual bool save_to_storage() = 0;
00194
00195         virtual bool load_from_storage() = 0;
00196
00197         protected:
00198             EventLog events[EVENT_BUFFER_SIZE];
00199             size_t eventCount;
00200             size_t writeIndex;
00201             mutex_t eventMutex;
00202             static uint16_t nextEventId;
00203             size_t eventsSinceFlush;
00204     };
00205

```

```

00245
00250 class EventManagerImpl : public EventManager {
00251     public:
00252         EventManagerImpl() {
00253             init(); // Safe to call virtual functions here
00254         }
00255
00256         public:
00257             bool save_to_storage() override {
00258                 if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00259                     bool status = fs_init();
00260                     if (!status) {
00261                         return false;
00262                     }
00263                 }
00264
00265                 FILE *file = fopen(EVENT_LOG_FILE, "a");
00266                 if (file) {
00267                     // Calculate start index for last EVENT_FLUSH_THRESHOLD events
00268                     size_t startIdx = (writeIndex >= eventsSinceFlush) ?
00269                         writeIndex - eventsSinceFlush :
00270                         EVENT_BUFFER_SIZE - (eventsSinceFlush - writeIndex);
00271
00272                     // Write only the most recent batch of events
00273                     for (size_t i = 0; i < eventsSinceFlush; i++) {
00274                         size_t idx = (startIdx + i) % EVENT_BUFFER_SIZE;
00275                         fprintf(file, "%u;%lu;%u;%u\n",
00276                             events[idx].id,
00277                             events[idx].timestamp,
00278                             events[idx].group,
00279                             events[idx].event
00280                         );
00281                     }
00282                     fclose(file);
00283                     uart_print("Events saved to storage", VerboseLevel::INFO);
00284                     return true;
00285                 }
00286             }
00287         return false;
00288     }
00289
00303     bool load_from_storage() override {
00304         // TODO: Implement based on chosen storage (SD/EEPROM)
00305         return false;
00306     }
00307 };
00308
00309
00313 extern EventManagerImpl eventManager;
00314
00319 class EventEmitter {
00320     public:
00321         template<typename T>
00322             static void emit(EventGroup group, T event) {
00323                 eventManager.log_event(
00324                     static_cast<uint8_t>(group),
00325                     static_cast<uint8_t>(event)
00326                 );
00327             }
00328 };
00329
00330 #endif // End of EventManagerGroup

```

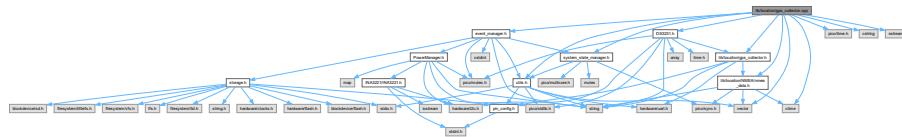
8.47 lib/location/gps_collector.cpp File Reference

```

#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

Functions

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

Variables

- NMEAData nmea data

8.47.1 Macro Definition Documentation

8.47.1.1 MAX RAW DATA LENGTH

```
#define MAX_RAW_DATA_LENGTH 256
```

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

8.47.2 Function Documentation

8.47.2.1 splitString()

```
std::vector< std::string > splitString (
```

Definition at line 18 of file [gps_collector.cpp](#).

Here is the caller graph for this function:

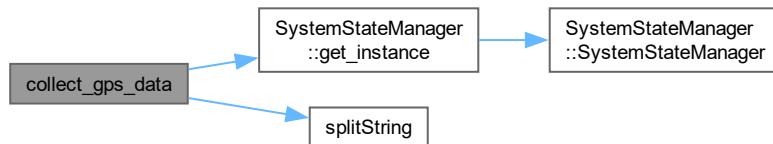


8.47.2.2 collect_gps_data()

```
void collect_gps_data ()
```

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

Here is the call graph for this function:



Here is the caller graph for this function:



8.47.3 Variable Documentation

8.47.3.1 nmea_data

`NMEAData nmea_data [extern]`

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

8.48 gps_collector.cpp

[Go to the documentation of this file.](#)

```

00001 // filepath: /c:/Users/Kuba/Desktop/inz/kubisat/software/kubisat_firmware/lib/GPS/gps_collector.cpp
00002 #include "lib/location/gps_collector.h"
00003 #include "utils.h"
00004 #include "pico/time.h"
00005 #include "lib/location/NMEA/nmea_data.h"
00006 #include "event_manager.h"
00007 #include <vector>
00008 #include <ctime>
00009 #include <cstring>
00010 #include "DS3231.h"
00011 #include <sstream>
00012 #include "system_state_manager.h"
00013
  
```

```

00014 #define MAX_RAW_DATA_LENGTH 256
00015
00016 extern NMEAData nmea_data;
00017
00018 std::vector<std::string> splitString(const std::string& str, char delimiter) {
00019     std::vector<std::string> tokens;
00020     std::stringstream ss(str);
00021     std::string token;
00022     while (std::getline(ss, token, delimiter)) {
00023         tokens.push_back(token);
00024     }
00025     return tokens;
00026 }
00027
00028 void collect_gps_data() {
00029
00030     if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
00031         return;
00032     }
00033
00034     static char raw_data_buffer[MAX_RAW_DATA_LENGTH];
00035     static int raw_data_index = 0;
00036
00037     while (uart_is_readable(GPS_UART_PORT)) {
00038         char c = uart_getc(GPS_UART_PORT);
00039
00040         if (c == '\r' || c == '\n') {
00041             // End of message
00042             if (raw_data_index > 0) {
00043                 raw_data_buffer[raw_data_index] = '\0';
00044                 std::string message(raw_data_buffer);
00045                 raw_data_index = 0;
00046
00047                 // Split the message into tokens
00048                 std::vector<std::string> tokens = splitString(message, ',');
00049
00050                 // Update the global vectors based on the sentence type
00051                 if (message.find("$GPRMC") == 0) {
00052                     nmea_data.update_rmc_tokens(tokens);
00053                 } else if (message.find("$GPGGA") == 0) {
00054                     nmea_data.update_gga_tokens(tokens);
00055                 }
00056             }
00057         } else {
00058             // Append to buffer
00059             if (raw_data_index < MAX_RAW_DATA_LENGTH - 1) {
00060                 raw_data_buffer[raw_data_index++] = c;
00061             } else {
00062                 raw_data_index = 0;
00063             }
00064         }
00065     }
00066 }

```

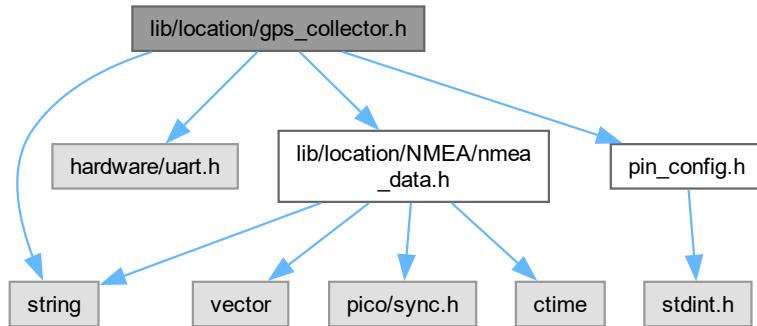
8.49 lib/location/gps_collector.h File Reference

```

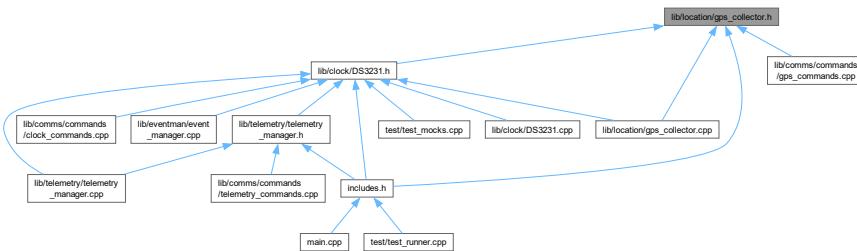
#include <string>
#include "hardware/uart.h"
#include "lib/location/NMEA/nmea_data.h"
#include "pin_config.h"

```

Include dependency graph for gps_collector.h:



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



Functions

- void [collect_gps_data \(\)](#)

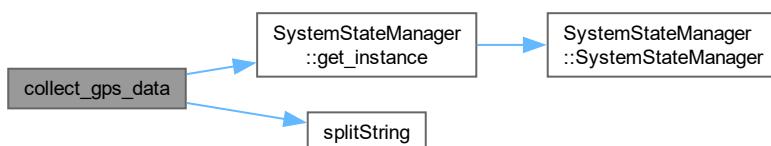
8.49.1 Function Documentation

8.49.1.1 collect_gps_data()

```
void collect_gps_data ()
```

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

Here is the call graph for this function:



Here is the caller graph for this function:



8.50 gps_collector.h

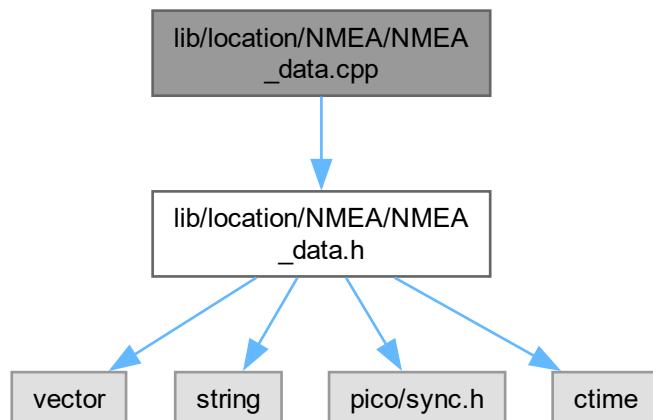
[Go to the documentation of this file.](#)

```

00001 #ifndef GPS_COLLECTOR_H
00002 #define GPS_COLLECTOR_H
00003
00004 #include <string>
00005 #include "hardware/uart.h"
00006 #include "lib/location/NMEA/nmea_data.h" // Include the new header
00007 #include "pin_config.h"
00008
00009 // Function to collect GPS data from the UART
00010 void collect_gps_data();
00011
00012 #endif
  
```

8.51 lib/location/NMEA/NMEA_data.cpp File Reference

```
#include "lib/location/NMEA/NMEA_data.h"
Include dependency graph for NMEA_data.cpp:
```



Variables

- `NMEAData nmea_data`

8.51.1 Variable Documentation

8.51.1.1 `nmea_data`

`NMEAData nmea_data`

Definition at line 3 of file `NMEA_data.cpp`.

8.52 NMEA_data.cpp

[Go to the documentation of this file.](#)

```

00001 #include "lib/location/NMEA/NMEA_data.h"
00002
00003 NMEAData nmea_data;
00004
00005 NMEAData::NMEAData() {
00006     mutex_init(&rmc_mutex_);
00007     mutex_init(&gga_mutex_);
00008 }
00009
0010 void NMEAData::update_rmc_tokens(const std::vector<std::string>& tokens) {
0011     mutex_enter_blocking(&rmc_mutex_);
0012     rmc_tokens_ = tokens;
0013     mutex_exit(&rmc_mutex_);
0014 }
0015
0016 void NMEAData::update_gga_tokens(const std::vector<std::string>& tokens) {
0017     mutex_enter_blocking(&gga_mutex_);
0018     gga_tokens_ = tokens;
0019     mutex_exit(&gga_mutex_);
0020 }
0021
0022 std::vector<std::string> NMEAData::get_rmc_tokens() const {
0023     mutex_enter_blocking(const_cast<mutex_t*>(&rmc_mutex_));
0024     std::vector<std::string> copy = rmc_tokens_;
0025     mutex_exit(const_cast<mutex_t*>(&rmc_mutex_));
0026     return copy;
0027 }
0028
0029 std::vector<std::string> NMEAData::get_gga_tokens() const {
0030     mutex_enter_blocking(const_cast<mutex_t*>(&gga_mutex_));
0031     std::vector<std::string> copy = gga_tokens_;
0032     mutex_exit(const_cast<mutex_t*>(&gga_mutex_));
0033     return copy;
0034 }
0035
0036 bool NMEAData::has_valid_time() const {
0037     return rmc_tokens_.size() >= 10 && rmc_tokens_[1].length() > 5;
0038 }
0039
0040 time_t NMEAData::get_unix_time() const {
0041     if (!has_valid_time()) {
0042         return 0; // Invalid time
0043     }
0044
0045     // Parse date and time from RMC tokens
0046     // Format: hhmmss.sss,A,ddmm.mmmm,N,dddmmyy,W,speed,course,ddmmyy
0047     std::string time_str = rmc_tokens_[1]; // hhmmss.sss
0048     std::string date_str = rmc_tokens_[9]; // ddmmyy
0049
0050     if (time_str.length() < 6 || date_str.length() < 6) {
0051         return 0;
0052     }
0053
0054     struct tm timeinfo = {0};
0055
0056     // Parse time: hours (0-1), minutes (2-3), seconds (4-5)
0057     timeinfo.tm_hour = std::stoi(time_str.substr(0, 2));

```

```

00058     timeinfo.tm_min = std::stoi(time_str.substr(2, 2));
00059     timeinfo.tm_sec = std::stoi(time_str.substr(4, 2));
00060
00061     // Parse date: day (0-1), month (2-3), year (4-5)
00062     timeinfo.tm_mday = std::stoi(date_str.substr(0, 2));
00063     timeinfo.tm_mon = std::stoi(date_str.substr(2, 2)) - 1; // Months from 0-11
00064     timeinfo.tm_year = std::stoi(date_str.substr(4, 2)) + 100; // Years since 1900
00065
00066     // Convert to unix time
00067     return mktime(&timeinfo);
00068 }

```

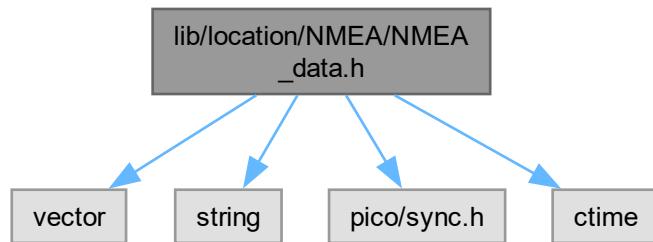
8.53 lib/location/NMEA/NMEA_data.h File Reference

```

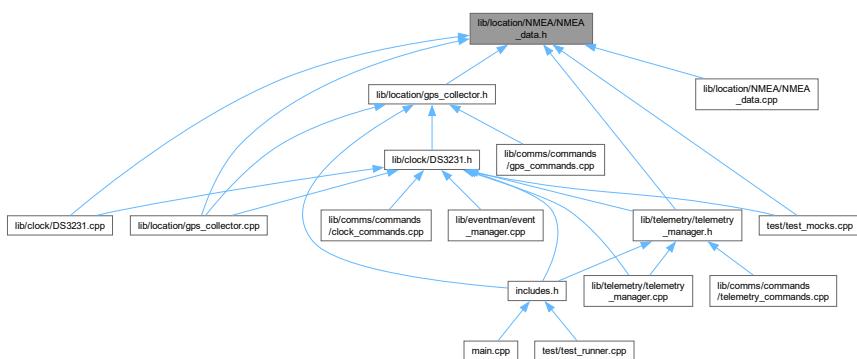
#include <vector>
#include <string>
#include "pico/sync.h"
#include <ctime>

```

Include dependency graph for NMEA_data.h:



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



Classes

- class [NMEAData](#)

Variables

- `NMEAData nmea_data`

8.53.1 Variable Documentation

8.53.1.1 nmea_data

`NMEAData nmea_data [extern]`

Definition at line 3 of file `NMEA_data.cpp`.

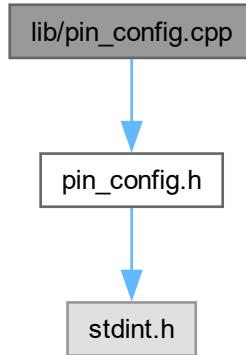
8.54 NMEA_data.h

Go to the documentation of this file.

```
00001 // filepath: /c:/Users/Kuba/Desktop/inz/kubisat/software/kubisat_firmware/lib/GPS/nmea_data.h
00002 #ifndef NMEA_DATA_H
00003 #define NMEA_DATA_H
00004
00005 #include <vector>
00006 #include <string>
00007 #include "pico/sync.h"
00008 #include <ctime>
00009
00010 class NMEAData {
00011 public:
00012     NMEAData();
00013     void update_rmc_tokens(const std::vector<std::string>& tokens);
00014     void update_gga_tokens(const std::vector<std::string>& tokens);
00015
00016     std::vector<std::string> get_rmc_tokens() const;
00017     std::vector<std::string> get_gga_tokens() const;
00018
00019     bool has_valid_time() const;
00020
00021     time_t get_unix_time() const;
00022
00023 private:
00024     std::vector<std::string> rmc_tokens_;
00025     std::vector<std::string> gga_tokens_;
00026     mutex_t rmc_mutex_;
00027     mutex_t gga_mutex_;
00028 };
00029
00030 extern NMEAData nmea_data;
00031
00032 #endif
```

8.55 lib/pin_config.cpp File Reference

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



Variables

- const int `lora_cs_pin` = 17
- const int `lora_reset_pin` = 22
- const int `lora_irq_pin` = 28
- uint8_t `lora_address_local` = 37
- uint8_t `lora_address_remote` = 21

8.55.1 Variable Documentation

8.55.1.1 lora_cs_pin

```
const int lora_cs_pin = 17
```

Definition at line 4 of file [pin_config.cpp](#).

8.55.1.2 lora_reset_pin

```
const int lora_reset_pin = 22
```

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

8.55.1.3 lora_irq_pin

```
const int lora_irq_pin = 28
```

Definition at line 6 of file [pin_config.cpp](#).

8.55.1.4 lora_address_local

```
uint8_t lora_address_local = 37
```

Definition at line 8 of file [pin_config.cpp](#).

8.55.1.5 lora_address_remote

```
uint8_t lora_address_remote = 21
```

Definition at line 9 of file [pin_config.cpp](#).

8.56 pin_config.cpp

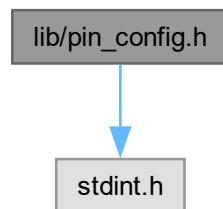
[Go to the documentation of this file.](#)

```
00001 #include "pin_config.h"
00002
00003 // LoRa constants
00004 const int lora_cs_pin = 17;           // LoRa radio chip select
00005 const int lora_reset_pin = 22;        // LoRa radio reset
00006 const int lora_irq_pin = 28;          // LoRa hardware interrupt pin
00007
00008 uint8_t lora_address_local = 37;      // address of this device
00009 uint8_t lora_address_remote = 21;
```

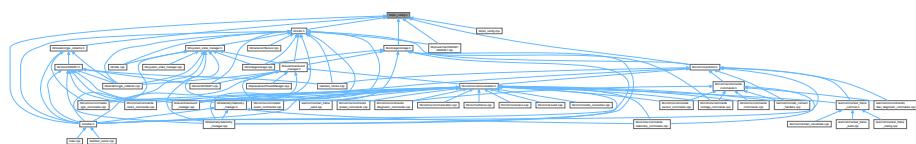
8.57 lib/pin_config.h File Reference

```
#include <stdint.h>
```

Include dependency graph for pin_config.h:



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



Macros

- `#define DEBUG_UART_PORT uart0`
- `#define DEBUG_UART_BAUD_RATE 115200`
- `#define DEBUG_UART_TX_PIN 0`
- `#define DEBUG_UART_RX_PIN 1`
- `#define MAIN_I2C_PORT i2c1`
- `#define MAIN_I2C_SDA_PIN 6`
- `#define MAIN_I2C_SCL_PIN 7`
- `#define GPS_UART_PORT uart1`
- `#define GPS_UART_BAUD_RATE 9600`
- `#define GPS_UART_TX_PIN 8`
- `#define GPS_UART_RX_PIN 9`
- `#define GPS_POWER_ENABLE_PIN 14`
- `#define BUFFER_SIZE 85`
- `#define SD_SPI_PORT spi1`
- `#define SD_MISO_PIN 12`
- `#define SD_MOSI_PIN 11`
- `#define SD_SCK_PIN 10`
- `#define SD_CS_PIN 13`
- `#define SD_CARD_DETECT_PIN 28`
- `#define SX1278_MISO 16`
- `#define SX1278_CS 17`
- `#define SX1278_SCK 18`
- `#define SX1278_MOSI 19`
- `#define SPI_PORT spi0`
- `#define READ_BIT 0x80`
- `#define LORA_DEFAULT_SPI spi0`
- `#define LORA_DEFAULT_SPI_FREQUENCY 8E6`
- `#define LORA_DEFAULT_SS_PIN 17`
- `#define LORA_DEFAULT_RESET_PIN 22`
- `#define LORA_DEFAULT_DIO0_PIN 20`
- `#define PA_OUTPUT_RFO_PIN 11`
- `#define PA_OUTPUT_PA_BOOST_PIN 12`

Variables

- `const int lora_cs_pin`
- `const int lora_reset_pin`
- `const int lora_irq_pin`
- `uint8_t lora_address_local`
- `uint8_t lora_address_remote`

8.57.1 Macro Definition Documentation

8.57.1.1 DEBUG_UART_PORT

```
#define DEBUG_UART_PORT uart0
```

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

8.57.1.2 DEBUG_UART_BAUD_RATE

```
#define DEBUG_UART_BAUD_RATE 115200
```

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

8.57.1.3 DEBUG_UART_TX_PIN

```
#define DEBUG_UART_TX_PIN 0
```

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

8.57.1.4 DEBUG_UART_RX_PIN

```
#define DEBUG_UART_RX_PIN 1
```

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

8.57.1.5 MAIN_I2C_PORT

```
#define MAIN_I2C_PORT i2c1
```

Definition at line [14](#) of file [pin_config.h](#).

8.57.1.6 MAIN_I2C_SDA_PIN

```
#define MAIN_I2C_SDA_PIN 6
```

Definition at line [15](#) of file [pin_config.h](#).

8.57.1.7 MAIN_I2C_SCL_PIN

```
#define MAIN_I2C_SCL_PIN 7
```

Definition at line [16](#) of file [pin_config.h](#).

8.57.1.8 GPS_UART_PORT

```
#define GPS_UART_PORT uart1
```

Definition at line [19](#) of file [pin_config.h](#).

8.57.1.9 GPS_UART_BAUD_RATE

```
#define GPS_UART_BAUD_RATE 9600
```

Definition at line [20](#) of file [pin_config.h](#).

8.57.1.10 GPS_UART_TX_PIN

```
#define GPS_UART_TX_PIN 8
```

Definition at line 21 of file [pin_config.h](#).

8.57.1.11 GPS_UART_RX_PIN

```
#define GPS_UART_RX_PIN 9
```

Definition at line 22 of file [pin_config.h](#).

8.57.1.12 GPS_POWER_ENABLE_PIN

```
#define GPS_POWER_ENABLE_PIN 14
```

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

8.57.1.13 BUFFER_SIZE

```
#define BUFFER_SIZE 85
```

Definition at line 25 of file [pin_config.h](#).

8.57.1.14 SD_SPI_PORT

```
#define SD_SPI_PORT spil
```

Definition at line 28 of file [pin_config.h](#).

8.57.1.15 SD_MISO_PIN

```
#define SD_MISO_PIN 12
```

Definition at line 29 of file [pin_config.h](#).

8.57.1.16 SD_MOSI_PIN

```
#define SD_MOSI_PIN 11
```

Definition at line 30 of file [pin_config.h](#).

8.57.1.17 SD_SCK_PIN

```
#define SD_SCK_PIN 10
```

Definition at line 31 of file [pin_config.h](#).

8.57.1.18 SD_CS_PIN

```
#define SD_CS_PIN 13
```

Definition at line 32 of file [pin_config.h](#).

8.57.1.19 SD_CARD_DETECT_PIN

```
#define SD_CARD_DETECT_PIN 28
```

Definition at line 33 of file [pin_config.h](#).

8.57.1.20 SX1278_MISO

```
#define SX1278_MISO 16
```

Definition at line 35 of file [pin_config.h](#).

8.57.1.21 SX1278_CS

```
#define SX1278_CS 17
```

Definition at line 36 of file [pin_config.h](#).

8.57.1.22 SX1278_SCK

```
#define SX1278_SCK 18
```

Definition at line 37 of file [pin_config.h](#).

8.57.1.23 SX1278_MOSI

```
#define SX1278_MOSI 19
```

Definition at line 38 of file [pin_config.h](#).

8.57.1.24 SPI_PORT

```
#define SPI_PORT spi0
```

Definition at line 40 of file [pin_config.h](#).

8.57.1.25 READ_BIT

```
#define READ_BIT 0x80
```

Definition at line 41 of file [pin_config.h](#).

8.57.1.26 LORA_DEFAULT_SPI

```
#define LORA_DEFAULT_SPI spi0
```

Definition at line 43 of file [pin_config.h](#).

8.57.1.27 LORA_DEFAULT_SPI_FREQUENCY

```
#define LORA_DEFAULT_SPI_FREQUENCY 8E6
```

Definition at line 44 of file [pin_config.h](#).

8.57.1.28 LORA_DEFAULT_SS_PIN

```
#define LORA_DEFAULT_SS_PIN 17
```

Definition at line 45 of file [pin_config.h](#).

8.57.1.29 LORA_DEFAULT_RESET_PIN

```
#define LORA_DEFAULT_RESET_PIN 22
```

Definition at line 46 of file [pin_config.h](#).

8.57.1.30 LORA_DEFAULT_DIO0_PIN

```
#define LORA_DEFAULT_DIO0_PIN 20
```

Definition at line 47 of file [pin_config.h](#).

8.57.1.31 PA_OUTPUT_RFO_PIN

```
#define PA_OUTPUT_RFO_PIN 11
```

Definition at line 49 of file [pin_config.h](#).

8.57.1.32 PA_OUTPUT_PA_BOOST_PIN

```
#define PA_OUTPUT_PA_BOOST_PIN 12
```

Definition at line 50 of file [pin_config.h](#).

8.57.2 Variable Documentation

8.57.2.1 lora_cs_pin

```
const int lora_cs_pin [extern]
```

Definition at line [4](#) of file [pin_config.cpp](#).

8.57.2.2 lora_reset_pin

```
const int lora_reset_pin [extern]
```

Definition at line [5](#) of file [pin_config.cpp](#).

8.57.2.3 lora_irq_pin

```
const int lora_irq_pin [extern]
```

Definition at line [6](#) of file [pin_config.cpp](#).

8.57.2.4 lora_address_local

```
uint8_t lora_address_local [extern]
```

Definition at line [8](#) of file [pin_config.cpp](#).

8.57.2.5 lora_address_remote

```
uint8_t lora_address_remote [extern]
```

Definition at line [9](#) of file [pin_config.cpp](#).

8.58 pin_config.h

[Go to the documentation of this file.](#)

```

00001 // pin_config.h
00002 #include <stdint.h>
00003
00004 #ifndef PIN_CONFIG_H
00005 #define PIN_CONFIG_H
00006
00007 //DEBUG uart
00008 #define DEBUG_UART_PORT uart0
00009 #define DEBUG_UART_BAUD_RATE 115200
00010
00011 #define DEBUG_UART_TX_PIN 0
00012 #define DEBUG_UART_RX_PIN 1
00013
00014 #define MAIN_I2C_PORT i2c1
00015 #define MAIN_I2C_SDA_PIN 6
00016 #define MAIN_I2C_SCL_PIN 7
00017
00018 // GPS configuration
00019 #define GPS_UART_PORT uart1
00020 #define GPS_UART_BAUD_RATE 9600
00021 #define GPS_UART_TX_PIN 8
00022 #define GPS_UART_RX_PIN 9
00023 #define GPS_POWER_ENABLE_PIN 14
00024
00025 #define BUFFER_SIZE 85 // NMEA sentences are usually under 85 chars
00026
00027 // SPI configuration for SD card
00028 #define SD_SPI_PORT spil
00029 #define SD_MISO_PIN 12
00030 #define SD_MOSI_PIN 11
00031 #define SD_SCK_PIN 10
00032 #define SD_CS_PIN 13
00033 #define SD_CARD_DETECT_PIN 28
00034
00035 #define SX1278_MISO 16
00036 #define SX1278_CS 17
00037 #define SX1278_SCK 18
00038 #define SX1278_MOSI 19
00039
00040 #define SPI_PORT spi0
00041 #define READ_BIT 0x80
00042
00043 #define LORA_DEFAULT_SPI spi0
00044 #define LORA_DEFAULT_SPI_FREQUENCY 8E6
00045 #define LORA_DEFAULT_SS_PIN 17
00046 #define LORA_DEFAULT_RESET_PIN 22
00047 #define LORA_DEFAULT_DIO0_PIN 20
00048
00049 #define PA_OUTPUT_RFO_PIN 11
00050 #define PA_OUTPUT_PA_BOOST_PIN 12
00051
00052
00053
00054 // LoRa constants - declare as extern
00055 extern const int lora_cs_pin; // LoRa radio chip select
00056 extern const int lora_reset_pin; // LoRa radio reset
00057 extern const int lora_irq_pin; // LoRa hardware interrupt pin
00058 extern uint8_t lora_address_local; // address of this device
00059 extern uint8_t lora_address_remote; // destination to send to
00060
00061
00062 #endif // PIN_CONFIG_H

```

8.59 lib/powerman/INA3221/INA3221.cpp File Reference

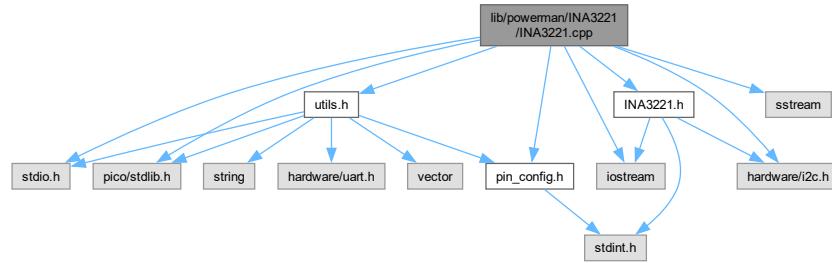
Implementation of the [INA3221](#) power monitor driver.

```

#include "INA3221.h"
#include <stdio.h>
#include "pico/stl.h"
#include "hardware/i2c.h"
#include <iostream>

```

```
#include "pin_config.h"
#include "utils.h"
#include <sstream>
Include dependency graph for INA3221.cpp:
```



8.59.1 Detailed Description

Implementation of the [INA3221](#) power monitor driver.

This file contains the implementation for the [INA3221](#) triple-channel power monitor, providing functionality for voltage, current, and power monitoring with alert capabilities.

Definition in file [INA3221.cpp](#).

8.60 INA3221.cpp

[Go to the documentation of this file.](#)

```

00001 #include "INA3221.h"
00002 #include <stdio.h>
00003 #include "pico/stlplib.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
00038
00039
00046 INA3221::INA3221(in3221_addr_t addr, i2c_inst_t* i2c)
00047     : _i2c_addr(addr), _i2c(i2c) {}
00048
00049
00056 bool INA3221::begin() {
00057     uart_print("INA3221 initializing...", VerboseLevel::DEBUG);
00058
00059     _shuntRes[0] = 10;
00060     _shuntRes[1] = 10;
00061     _shuntRes[2] = 10;
00062
00063     _filterRes[0] = 10;
00064     _filterRes[1] = 10;
00065     _filterRes[2] = 10;
00066
00067     uint16_t manuf_id = get_manufacturer_id();
00068     uint16_t die_id = get_die_id();
00069     std::stringstream ss;
00070     ss << "INA3221 Manufacturer ID: 0x" << std::hex << manuf_id
00071             << ", Die ID: 0x" << die_id;
00072     uart_print(ss.str(), VerboseLevel::INFO);
00073 }
```

```

00073
00074     if (manuf_id == 0x5449 && die_id == 0x3220) {
00075         uart_print("INA3221 found and initialized.", VerbosityLevel::DEBUG);
00076         return true;
00077     } else {
00078         uart_print("INA3221 initialization failed. Incorrect IDs.", VerbosityLevel::ERROR);
00079         return false;
00080     }
00081
00082 }
00083
00084
00085 void INA3221::reset(){
00086     conf_reg_t conf_reg;
00087
00088     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00089     conf_reg.reset = 1;
00090     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00091 }
00092
00093
00094 uint16_t INA3221::get_manufacturer_id() {
00095     uint16_t id = 0;
00096     _read(INA3221_REG_MANUF_ID, &id);
00097     return id;
00098 }
00099
00100
00101 uint16_t INA3221::get_die_id() {
00102     uint16_t id = 0;
00103     _read(INA3221_REG_DIE_ID, &id);
00104     return id;
00105 }
00106
00107
00108 }
00109
00110
00111
00112
00113 uint16_t INA3221::read_register(ina3221_reg_t reg){
00114     uint16_t val = 0;
00115     _read(reg, &val);
00116     return val;
00117 }
00118
00119
00120 }
00121
00122
00123
00124 //configure
00125
00126
00127 void INA3221::set_mode_power_down(){
00128     conf_reg_t conf_reg;
00129
00130     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00131     conf_reg.mode_bus_en = 0;
00132     conf_reg.mode_continious_en =0 ;
00133     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00134 }
00135
00136
00137
00138 void INA3221::set_mode_continuous(){
00139     conf_reg_t conf_reg;
00140
00141     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00142     conf_reg.mode_continious_en =1;
00143     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00144 }
00145
00146
00147
00148 void INA3221::set_mode_triggered(){
00149     conf_reg_t conf_reg;
00150
00151     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00152     conf_reg.mode_continious_en = 0;
00153     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00154 }
00155
00156
00157
00158 void INA3221::set_shunt_measurement_enable(){
00159     conf_reg_t conf_reg;
00160
00161     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00162     conf_reg.mode_shunt_en = 1;
00163     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00164 }
00165
00166
00167
00168 void INA3221::set_shunt_measurement_disable(){
00169     conf_reg_t conf_reg;
00170
00171     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00172     conf_reg.mode_shunt_en = 0;
00173     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00174 }
00175
00176
00177
00178
00179
00180
00181
00182
00183
00184
00185
00186
00187
00188
00189
00190
00191
00192
00193
00194
00195
00196
00197
00198
00199
00200
00201
00202
00203

```

```
00204 }
00205
00206
00211 void INA3221::set_bus_measurement_enable() {
00212     conf_reg_t conf_reg;
00213
00214     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00215     conf_reg.mode_bus_en = 1;
00216     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00217 }
00218
00219
00224 void INA3221::set_bus_measurement_disable() {
00225     conf_reg_t conf_reg;
00226
00227     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00228     conf_reg.mode_bus_en = 0;
00229     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00230 }
00231
00232
00238 void INA3221::set_averaging_mode(ina3221_avg_mode_t mode) {
00239     conf_reg_t conf_reg;
00240
00241     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00242     conf_reg.avg_mode = mode;
00243     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00244 }
00245
00246
00252 void INA3221::set_bus_conversion_time(ina3221_conv_time_t convTime) {
00253     conf_reg_t conf_reg;
00254
00255     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00256     conf_reg.bus_conv_time = convTime;
00257     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00258 }
00259
00260
00266 void INA3221::set_shunt_conversion_time(ina3221_conv_time_t convTime) {
00267     conf_reg_t conf_reg;
00268
00269     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00270     conf_reg.shunt_conv_time = convTime;
00271     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00272 }
00273
00274
00275 //get measurement
00282 int32_t INA3221::get_shunt_voltage(ina3221_ch_t channel) {
00283     int32_t res;
00284     ina3221_reg_t reg;
00285     uint16_t val_raw = 0;
00286
00287     switch(channel){
00288         case INA3221_CH1:
00289             reg = INA3221_REG_CH1_SHUNTV;
00290             break;
00291         case INA3221_CH2:
00292             reg = INA3221_REG_CH2_SHUNTV;
00293             break;
00294         case INA3221_CH3:
00295             reg = INA3221_REG_CH3_SHUNTV;
00296             break;
00297     }
00298
00299     _read(reg, &val_raw);
00300
00301     res = (int16_t) (val_raw >> 3);
00302     res *= SHUNT_VOLTAGE_LSB_UV;
00303
00304     return res;
00305 }
00306
00307
00314 float INA3221::get_current_ma(ina3221_ch_t channel) {
00315     int32_t shunt_uV = 0;
00316     float current_A = 0;
00317
00318     shunt_uV = get_shunt_voltage(channel);
00319     current_A = shunt_uV / (int32_t)_shuntRes[channel];
00320     return current_A;
00321 }
00322
00323
00330 float INA3221::get_voltage(ina3221_ch_t channel) {
00331     float voltage_V = 0.0;
```

```

00332     ina3221_reg_t reg;
00333     uint16_t val_raw = 0;
00334
00335     switch(channel){
00336         case INA3221_CH1:
00337             reg = INA3221_REG_CH1_BUSV;
00338             break;
00339         case INA3221_CH2:
00340             reg = INA3221_REG_CH2_BUSV;
00341             break;
00342         case INA3221_CH3:
00343             reg = INA3221_REG_CH3_BUSV;
00344             break;
00345     }
00346
00347     _read(reg, &val_raw);
00348     voltage_V = val_raw / 1000.0;
00349     return voltage_V;
00350 }
00351
00352
00353 // alerts
00354 void INA3221::set_warn_alert_limit(ina3221_ch_t channel, float voltage_v) {
00355     ina3221_reg_t reg;
00356     uint16_t val = (uint16_t)(voltage_v * 1000); // Convert V to mV
00357
00358     switch(channel) {
00359         case INA3221_CH1:
00360             reg = INA3221_REG_CH1_WARNING_ALERT_LIM;
00361             break;
00362         case INA3221_CH2:
00363             reg = INA3221_REG_CH2_WARNING_ALERT_LIM;
00364             break;
00365         case INA3221_CH3:
00366             reg = INA3221_REG_CH3_WARNING_ALERT_LIM;
00367             break;
00368     }
00369     _write(reg, &val);
00370 }
00371
00372
00373
00374 void INA3221::set_crit_alert_limit(ina3221_ch_t channel, float voltage_v) {
00375     ina3221_reg_t reg;
00376     uint16_t val = (uint16_t)(voltage_v * 1000); // Convert V to mV
00377
00378     switch(channel) {
00379         case INA3221_CH1:
00380             reg = INA3221_REG_CH1_CRIT_ALERT_LIM;
00381             break;
00382         case INA3221_CH2:
00383             reg = INA3221_REG_CH2_CRIT_ALERT_LIM;
00384             break;
00385         case INA3221_CH3:
00386             reg = INA3221_REG_CH3_CRIT_ALERT_LIM;
00387             break;
00388     }
00389     _write(reg, &val);
00400 }
00401
00402
00403
00404 void INA3221::set_power_valid_limit(float voltage_upper_v, float voltage_lower_v) {
00405     uint16_t val;
00406
00407     val = (uint16_t)(voltage_upper_v * 1000);
00408     _write(INA3221_REG_PWR_VALID_HI_LIM, &val);
00409
00410     val = (uint16_t)(voltage_lower_v * 1000);
00411     _write(INA3221_REG_PWR_VALID_LO_LIM, &val);
00412 }
00413
00414
00415
00416
00417
00418
00419
00420
00421
00422
00423
00424
00425
00426 void INA3221::enable_alerts() {
00427     masken_reg_t masken;
00428     _read(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00429
00430     masken.warn_alert_ch1 = 1;
00431     masken.warn_alert_ch2 = 1;
00432     masken.warn_alert_ch3 = 1;
00433     masken.crit_alert_ch1 = 1;
00434     masken.crit_alert_ch2 = 1;
00435     masken.crit_alert_ch3 = 1;
00436     masken.pwr_valid_alert = 1;
00437
00438     _write(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00439 }
00440
00441

```

```

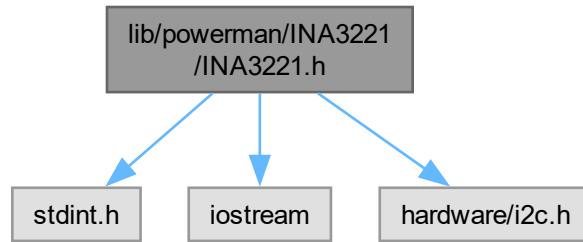
00448 bool INA3221::get_warn_alert(ina3221_ch_t channel) {
00449     masken_reg_t masken;
00450     _read(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00451
00452     switch(channel) {
00453         case INA3221_CH1: return masken.warn_alert_ch1;
00454         case INA3221_CH2: return masken.warn_alert_ch2;
00455         case INA3221_CH3: return masken.warn_alert_ch3;
00456         default: return false;
00457     }
00458 }
00459
00460
00461 bool INA3221::get_crit_alert(ina3221_ch_t channel) {
00462     masken_reg_t masken;
00463     _read(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00464
00465     switch(channel) {
00466         case INA3221_CH1: return masken.crit_alert_ch1;
00467         case INA3221_CH2: return masken.crit_alert_ch2;
00468         case INA3221_CH3: return masken.crit_alert_ch3;
00469         default: return false;
00470     }
00471 }
00472
00473
00474
00475 bool INA3221::get_power_valid_alert() {
00476     masken_reg_t masken;
00477     _read(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00478     return masken.pwr_valid_alert;
00479 }
00480
00481
00482
00483 void INA3221::set_alert_latch(bool enable) {
00484     masken_reg_t masken;
00485     _read(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00486     masken.warn_alert_latch_en = enable;
00487     masken.crit_alert_latch_en = enable;
00488     _write(INA3221_REG_MASK_ENABLE, (uint16_t*)&masken);
00489 }
00490
00491
00492
00493
00494 // private
00495 void INA3221::_read(ina3221_reg_t reg, uint16_t *val) {
00496     uint8_t reg_buf = reg;
00497     uint8_t data[2];
00498
00499     int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, &reg_buf, 1, true);
00500     if (ret != 1) {
00501         std::cerr << "Failed to write register address to I2C device." << std::endl;
00502         return;
00503     }
00504
00505
00506     ret = i2c_read_blocking(MAIN_I2C_PORT, _i2c_addr, data, 2, false);
00507     if (ret != 2) {
00508         std::cerr << "Failed to read data from I2C device." << std::endl;
00509         return;
00510     }
00511
00512     *val = (data[0] << 8) | data[1];
00513 }
00514
00515
00516
00517
00518
00519
00520
00521
00522
00523
00524
00525
00526
00527
00528
00529
00530
00531
00532
00533
00534
00535
00536
00537
00538
00539 void INA3221::_write(ina3221_reg_t reg, uint16_t *val) {
00540     uint8_t buf[3];
00541     buf[0] = reg;
00542     buf[1] = (*val >> 8) & 0xFF; // MSB
00543     buf[2] = (*val) & 0xFF; // LSB
00544
00545     int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, buf, 3, false);
00546     if (ret != 3) {
00547         std::cerr << "Failed to write data to I2C device." << std::endl;
00548     }
00549 }
```

8.61 lib/powerman/INA3221/INA3221.h File Reference

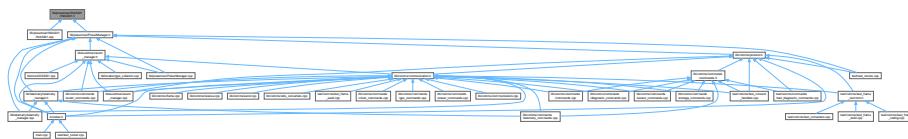
Header file for the [INA3221](#) triple-channel power monitor driver.

```
#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_ALEP` ,
`INA3221_REG_CH1_WARNING_ALERT_LIM` , `INA3221_REG_CH2_CRIT_ALERT_LIM` , `INA3221_REG_CH2_WARNING_A` ,
`INA3221_REG_CH3_CRIT_ALERT_LIM` , `INA3221_REG_CH3_WARNING_ALERT_LIM` , `INA3221_REG_SHUNTV_SUM` , `INA3221_REG_SHUNTV_SUM_LIM` ,
`INA3221_REG_MASK_ENABLE` , `INA3221_REG_PWR_VALID_HI_LIM` , `INA3221_REG_PWR_VALID_LO_LIM` , `INA3221_REG_MANUF_ID` = 0xFE , `INA3221_REG_DIE_ID` = 0xFF }
- Register addresses for [INA3221](#).*

- enum `ina3221_conv_time_t` {
 `INA3221_REG_CONF_CT_140US` = 0 , `INA3221_REG_CONF_CT_204US` , `INA3221_REG_CONF_CT_332US` , `INA3221_REG_CONF_CT_588US` ,
 `INA3221_REG_CONF_CT_1100US` , `INA3221_REG_CONF_CT_2116US` , `INA3221_REG_CONF_CT_4156US` ,
 `INA3221_REG_CONF_CT_8244US` }
- Conversion time settings.*
- enum `ina3221_avg_mode_t` {
 `INA3221_REG_CONF_AVG_1` = 0 , `INA3221_REG_CONF_AVG_4` , `INA3221_REG_CONF_AVG_16` ,
 `INA3221_REG_CONF_AVG_64` ,
 `INA3221_REG_CONF_AVG_128` , `INA3221_REG_CONF_AVG_256` , `INA3221_REG_CONF_AVG_512` ,
 `INA3221_REG_CONF_AVG_1024` }
- Averaging mode settings.*

Variables

- const int `INA3221_CH_NUM` = 3
Number of channels in `INA3221`.
- const int `SHUNT_VOLTAGE_LSB_UV` = 5
LSB value for shunt voltage measurements in microvolts.

8.61.1 Detailed Description

Header file for the `INA3221` triple-channel power monitor driver.

Definition in file `INA3221.h`.

8.61.2 Enumeration Type Documentation

8.61.2.1 `ina3221_addr_t`

`enum ina3221_addr_t`

Enumerator

<code>INA3221_ADDR40_GND</code>	
<code>INA3221_ADDR41_VCC</code>	
<code>INA3221_ADDR42_SDA</code>	
<code>INA3221_ADDR43_SCL</code>	

Definition at line 12 of file `INA3221.h`.

8.61.2.2 `ina3221_ch_t`

`enum ina3221_ch_t`

Enumerator

<code>INA3221_CH1</code>	
<code>INA3221_CH2</code>	
<code>INA3221_CH3</code>	

Definition at line 23 of file `INA3221.h`.

8.61.2.3 ina3221_reg_t

enum `ina3221_reg_t`

Register addresses for [INA3221](#).

Enumerator

INA3221_REG_CONF
INA3221_REG_CH1_SHUNTV
INA3221_REG_CH1_BUSV
INA3221_REG_CH2_SHUNTV
INA3221_REG_CH2_BUSV
INA3221_REG_CH3_SHUNTV
INA3221_REG_CH3_BUSV
INA3221_REG_CH1_CRIT_ALERT_LIM
INA3221_REG_CH1_WARNING_ALERT_LIM
INA3221_REG_CH2_CRIT_ALERT_LIM
INA3221_REG_CH2_WARNING_ALERT_LIM
INA3221_REG_CH3_CRIT_ALERT_LIM
INA3221_REG_CH3_WARNING_ALERT_LIM
INA3221_REG_SHUNTV_SUM
INA3221_REG_SHUNTV_SUM_LIM
INA3221_REG_MASK_ENABLE
INA3221_REG_PWR_VALID_HI_LIM
INA3221_REG_PWR_VALID_LO_LIM
INA3221_REG_MANUF_ID
INA3221_REG_DIE_ID

Definition at line 38 of file [INA3221.h](#).

8.61.2.4 ina3221_conv_time_t

enum `ina3221_conv_time_t`

Conversion time settings.

Time taken for each measurement conversion

Enumerator

INA3221_REG_CONF_CT_140US
INA3221_REG_CONF_CT_204US
INA3221_REG_CONF_CT_332US
INA3221_REG_CONF_CT_588US
INA3221_REG_CONF_CT_1100US
INA3221_REG_CONF_CT_2116US
INA3221_REG_CONF_CT_4156US
INA3221_REG_CONF_CT_8244US

Definition at line 65 of file [INA3221.h](#).

8.61.2.5 ina3221_avg_mode_t

enum [ina3221_avg_mode_t](#)

Averaging mode settings.

Number of samples to average for each measurement

Enumerator

INA3221_REG_CONF_AVG_1	
INA3221_REG_CONF_AVG_4	
INA3221_REG_CONF_AVG_16	
INA3221_REG_CONF_AVG_64	
INA3221_REG_CONF_AVG_128	
INA3221_REG_CONF_AVG_256	
INA3221_REG_CONF_AVG_512	
INA3221_REG_CONF_AVG_1024	

Definition at line 80 of file [INA3221.h](#).

8.61.3 Variable Documentation

8.61.3.1 INA3221_CH_NUM

const int INA3221_CH_NUM = 3

Number of channels in [INA3221](#).

Definition at line 30 of file [INA3221.h](#).

8.61.3.2 SHUNT_VOLTAGE_LSB_UV

const int SHUNT_VOLTAGE_LSB_UV = 5

LSB value for shunt voltage measurements in microvolts.

Definition at line 32 of file [INA3221.h](#).

8.62 INA3221.h

[Go to the documentation of this file.](#)

```

00001 #ifndef BEASTDEVICES_INA3221_H
00002 #define BEASTDEVICES_INA3221_H
00003
00004 #include <stdint.h>
00005 #include <iostream>
00006 #include <hardware/i2c.h>
00007
00012 typedef enum {
00013     INA3221_ADDR40_GND = 0b1000000, // A0 pin -> GND
00014     INA3221_ADDR41_VCC = 0b1000001, // A0 pin -> VCC
00015     INA3221_ADDR42_SDA = 0b1000010, // A0 pin -> SDA
00016     INA3221_ADDR43_SCL = 0b1000011 // A0 pin -> SCL
00017 } ina3221_addr_t;
00018
00023 typedef enum {
00024     INA3221_CH1 = 0,
00025     INA3221_CH2,
00026     INA3221_CH3,
00027 } ina3221_ch_t;
00028
00030 const int INA3221_CH_NUM = 3;
00032 const int SHUNT_VOLTAGE_LSB_UV = 5;
00033
00034
00038 typedef enum {
00039     INA3221_REG_CONF = 0,
00040     INA3221_REG_CH1_SHUNTV,
00041     INA3221_REG_CH1_BUSV,
00042     INA3221_REG_CH2_SHUNTV,
00043     INA3221_REG_CH2_BUSV,
00044     INA3221_REG_CH3_SHUNTV,
00045     INA3221_REG_CH3_BUSV,
00046     INA3221_REG_CH1_CRIT_ALERT_LIM,
00047     INA3221_REG_CH1_WARNING_ALERT_LIM,
00048     INA3221_REG_CH2_CRIT_ALERT_LIM,
00049     INA3221_REG_CH2_WARNING_ALERT_LIM,
00050     INA3221_REG_CH3_CRIT_ALERT_LIM,
00051     INA3221_REG_CH3_WARNING_ALERT_LIM,
00052     INA3221_REG_SHUNTV_SUM,
00053     INA3221_REG_SHUNTV_SUM_LIM,
00054     INA3221_REG_MASK_ENABLE,
00055     INA3221_REG_PWR_VALID_HI_LIM,
00056     INA3221_REG_PWR_VALID_LO_LIM,
00057     INA3221_REG_MANUF_ID = 0xFE,
00058     INA3221_REG_DIE_ID = 0xFF
00059 } ina3221_reg_t;
00060
00065 typedef enum {
00066     INA3221_REG_CONF_CT_140US = 0,
00067     INA3221_REG_CONF_CT_204US,
00068     INA3221_REG_CONF_CT_332US,
00069     INA3221_REG_CONF_CT_588US,
00070     INA3221_REG_CONF_CT_1100US,
00071     INA3221_REG_CONF_CT_2116US,
00072     INA3221_REG_CONF_CT_4156US,
00073     INA3221_REG_CONF_CT_8244US
00074 } ina3221_conv_time_t;
00075
00080 typedef enum {
00081     INA3221_REG_CONF_AVG_1 = 0,
00082     INA3221_REG_CONF_AVG_4,
00083     INA3221_REG_CONF_AVG_16,
00084     INA3221_REG_CONF_AVG_64,
00085     INA3221_REG_CONF_AVG_128,
00086     INA3221_REG_CONF_AVG_256,
00087     INA3221_REG_CONF_AVG_512,
00088     INA3221_REG_CONF_AVG_1024
00089 } ina3221_avg_mode_t;
00090
00096 class INA3221 {
00097
00101     typedef struct {
00102         uint16_t mode_shunt_en:1;
00103         uint16_t mode_bus_en:1;
00104         uint16_t mode_continious_en:1;
00105         uint16_t shunt_conv_time:3;
00106         uint16_t bus_conv_time:3;
00107         uint16_t avg_mode:3;
00108         uint16_t ch3_en:1;
00109         uint16_t ch2_en:1;
00110         uint16_t ch1_en:1;
00111         uint16_t reset:1;

```

```
00112     } conf_reg_t;
00113
00117     typedef struct {
00118         uint16_t conv_ready:1;
00119         uint16_t timing_ctrl_alert:1;
00120         uint16_t pwr_valid_alert:1;
00121         uint16_t warn_alert_ch3:1;
00122         uint16_t warn_alert_ch2:1;
00123         uint16_t warn_alert_ch1:1;
00124         uint16_t shunt_sum_alert:1;
00125         uint16_t crit_alert_ch3:1;
00126         uint16_t crit_alert_ch2:1;
00127         uint16_t crit_alert_ch1:1;
00128         uint16_t crit_alert_latch_en:1;
00129         uint16_t warn_alert_latch_en:1;
00130         uint16_t shunt_sum_en_ch3:1;
00131         uint16_t shunt_sum_en_ch2:1;
00132         uint16_t shunt_sum_en_ch1:1;
00133         uint16_t reserved:1;
00134     } masken_reg_t;
00135
00136     // I2C address
00137     ina3221_addr_t _i2c_addr;
00138     i2c_inst_t* _i2c;
00139
00140     // Shunt resistance in mOhm
00141     uint32_t _shuntRes[INA3221_CH_NUM];
00142
00143     // Series filter resistance in Ohm
00144     uint32_t _filterRes[INA3221_CH_NUM];
00145
00146     // Value of Mask/Enable register.
00147     masken_reg_t _masken_reg;
00148
00149     // Reads 16 bytes from a register.
00150     void _read(ina3221_reg_t reg, uint16_t *val);
00151
00152     // Writes 16 bytes to a register.
00153     void _write(ina3221_reg_t reg, uint16_t *val);
00154
00155 public:
00156
00157     INA3221(ina3221_addr_t addr, i2c_inst_t* i2c);
00158     // Initializes INA3221
00159     bool begin();
00160
00161     // Gets a register value.
00162     uint16_t read_register(ina3221_reg_t reg);
00163
00164     // Resets INA3221
00165     void reset();
00166
00167     // Sets operating mode to power-down
00168     void set_mode_power_down();
00169
00170     // Sets operating mode to continuous
00171     void set_mode_continuous();
00172
00173     // Sets operating mode to triggered (single-shot)
00174     void set_mode_triggered();
00175
00176     // Enables shunt-voltage measurement
00177     void set_shunt_measurement_enable();
00178
00179     // Disables shunt-voltage measurement
00180     void set_shunt_measurement_disable();
00181
00182     // Enables bus-voltage measurement
00183     void set_bus_measurement_enable();
00184
00185     // Disables bus-voltage measurement
00186     void set_bus_measurement_disable();
00187
00188     // Sets averaging mode. Sets number of samples that are collected
00189     // and averaged together.
00190     void set_averaging_mode(ina3221_avg_mode_t mode);
00191
00192     // Sets bus-voltage conversion time.
00193     void set_bus_conversion_time(ina3221_conv_time_t convTime);
00194
00195     // Sets shunt-voltage conversion time.
00196     void set_shunt_conversion_time(ina3221_conv_time_t convTime);
00197
00198     // Gets manufacturer ID.
00199     // Should read 0x5449.
00200     uint16_t get_manufacturer_id();
```

```

00202 // Gets die ID.
00203 // Should read 0x3220.
00204 uint16_t get_die_id();
00205
00206 // Gets shunt voltage in uV.
00207 int32_t get_shunt_voltage(ina3221_ch_t channel);
00208
00209 // Gets current in A.
00210 float get_current(ina3221_ch_t channel);
00211
00212 float get_current_ma(ina3221_ch_t channel);
00213
00214 // Gets bus voltage in V.
00215 float get_voltage(ina3221_ch_t channel);
00216
00217 void set_warn_alert_limit(ina3221_ch_t channel, float voltage_v);
00218 void set_crit_alert_limit(ina3221_ch_t channel, float voltage_v);
00219 void set_power_valid_limit(float voltage_upper_v, float voltage_lower_v);
00220 void enable_alerts();
00221 bool get_warn_alert(ina3221_ch_t channel);
00222 bool get_crit_alert(ina3221_ch_t channel);
00223 bool get_power_valid_alert();
00224 void set_alert_latch(bool enable);
00225 };
00226
00227 #endif

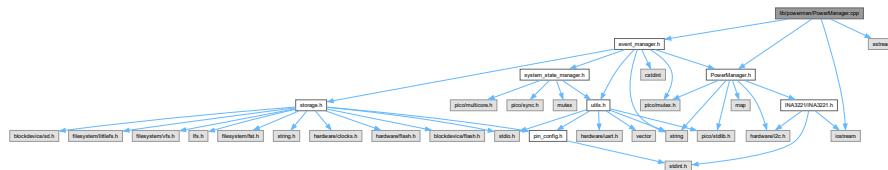
```

8.63 lib/powerman/PowerManager.cpp File Reference

```

#include "PowerManager.h"
#include <iostream>
#include <sstream>
#include "event_manager.h"
Include dependency graph for PowerManager.cpp:

```



8.64 PowerManager.cpp

Go to the documentation of this file.

```

00001 #include "PowerManager.h"
00002 #include <iostream>
00003 #include <sstream>
00004 #include "event_manager.h"
00005
00006 PowerManager::PowerManager(i2c_inst_t* i2c)
00007     : ina3221_(INA3221_ADDR40_GND, i2c) {
00008     recursive_mutex_init(&powerman_mutex_);
00009 }
0010
0011     bool PowerManager::initialize() {
0012         recursive_mutex_enter_blocking(&powerman_mutex_);
0013         initialized_ = ina3221_.begin();
0014
0015         if (initialized_) {
0016             // Set up alerts
0017             ina3221_.set_warn_alert_limit(INA3221_CH2, VOLTAGE_LOW_THRESHOLD);
0018             ina3221_.set_crit_alert_limit(INA3221_CH2, VOLTAGE_OVERCHARGE_THRESHOLD);
0019             ina3221_.set_power_valid_limit(VOLTAGE_OVERCHARGE_THRESHOLD, VOLTAGE_LOW_THRESHOLD);
0020             ina3221_.enable_alerts();
0021             ina3221_.set_alert_latch(true);
0022         }

```

```
00023     recursive_mutex_exit(&powerman_mutex_);
00024     return initialized_;
00025 }
00026
00027 std::string PowerManager::read_device_ids() {
00028     if (!initialized_) return "noinit";
00029     recursive_mutex_enter_blocking(&powerman_mutex_);
00030     std::stringstream man_ss;
00031     man_ss << std::hex << ina3221_.get_manufacturer_id();
00032     std::string MAN = "MAN 0x" + man_ss.str();
00033
00034     std::stringstream die_ss;
00035     die_ss << std::hex << ina3221_.get_die_id();
00036     std::string DIE = "DIE 0x" + die_ss.str();
00037     recursive_mutex_exit(&powerman_mutex_);
00038     return MAN + " - " + DIE;
00039 }
00040
00041
00042 float PowerManager::get_voltage_battery() {
00043     if (!initialized_) return 0.0f;
00044     recursive_mutex_enter_blocking(&powerman_mutex_);
00045     float voltage = ina3221_.get_voltage(INA3221_CH1);
00046     recursive_mutex_exit(&powerman_mutex_);
00047     return voltage;
00048 }
00049
00050 float PowerManager::get_voltage_5v() {
00051     if (!initialized_) return 0.0f;
00052     recursive_mutex_enter_blocking(&powerman_mutex_);
00053     float voltage = ina3221_.get_voltage(INA3221_CH2);
00054     recursive_mutex_exit(&powerman_mutex_);
00055     return voltage;
00056 }
00057
00058 float PowerManager::get_current_charge_usb() {
00059     if (!initialized_) return 0.0f;
00060     recursive_mutex_enter_blocking(&powerman_mutex_);
00061     float current = ina3221_.get_current_ma(INA3221_CH1);
00062     recursive_mutex_exit(&powerman_mutex_);
00063     return current;
00064 }
00065
00066 float PowerManager::get_current_draw() {
00067     if (!initialized_) return 0.0f;
00068     recursive_mutex_enter_blocking(&powerman_mutex_);
00069     float current = ina3221_.get_current_ma(INA3221_CH2);
00070     recursive_mutex_exit(&powerman_mutex_);
00071     return current;
00072 }
00073
00074 float PowerManager::get_current_charge_solar() {
00075     if (!initialized_) return 0.0f;
00076     recursive_mutex_enter_blocking(&powerman_mutex_);
00077     float current = ina3221_.get_current_ma(INA3221_CH3);
00078     recursive_mutex_exit(&powerman_mutex_);
00079     return current;
00080 }
00081
00082 float PowerManager::get_current_charge_total() {
00083     if (!initialized_) return 0.0f;
00084     recursive_mutex_enter_blocking(&powerman_mutex_);
00085     float current = ina3221_.get_current_ma(INA3221_CH1) + ina3221_.get_current_ma(INA3221_CH3);
00086     recursive_mutex_exit(&powerman_mutex_);
00087     return current;
00088 }
00089
00090 void PowerManager::configure(const std::map<std::string, std::string>& config) {
00091     if (!initialized_) return;
00092     recursive_mutex_enter_blocking(&powerman_mutex_);
00093
00094     if (config.find("operating_mode") != config.end()) {
00095         if (config.at("operating_mode") == "continuous") {
00096             ina3221_.set_mode_continuous();
00097         }
00098     }
00099
00100    if (config.find("averaging_mode") != config.end()) {
00101        int avg_mode = std::stoi(config.at("averaging_mode"));
00102        switch(avg_mode) {
00103            case 1:
00104                ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_1);
00105                break;
00106            case 4:
00107                ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_4);
00108                break;
00109            case 16:
```

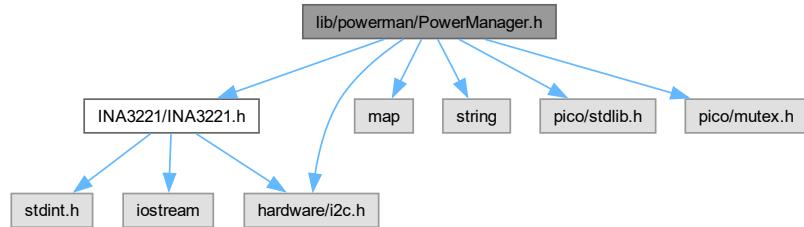
```

00110         ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_16);
00111         break;
00112     default:
00113         ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_16);
00114     }
00115 }
00116 recursive_mutex_exit(&powerman_mutex_);
00117 }
00118
00119 bool PowerManager::is_charging_solar() {
00120     if (!initialized_) return false;
00121     recursive_mutex_enter_blocking(&powerman_mutex_);
00122     bool active = get_current_charge_solar() > SOLAR_CURRENT_THRESHOLD;
00123     recursive_mutex_exit(&powerman_mutex_);
00124     return active;
00125 }
00126
00127 bool PowerManager::is_charging_usb() {
00128     if (!initialized_) return false;
00129     recursive_mutex_enter_blocking(&powerman_mutex_);
00130     bool connected = get_current_charge_usb() > USB_CURRENT_THRESHOLD;
00131     recursive_mutex_exit(&powerman_mutex_);
00132     return connected;
00133 }
00134
00135 bool PowerManager::check_power_alerts() {
00136     if (!initialized_) return false;
00137     recursive_mutex_enter_blocking(&powerman_mutex_);
00138
00139     bool status_changed = false;
00140
00141     // Check warning alert (low battery)
00142     if (ina3221_.get_warn_alert(INA3221_CH2)) {
00143         EventEmitter::emit(EventGroup::POWER, PowerEvent::LOW_BATTERY);
00144         status_changed = true;
00145     }
00146
00147     // Check critical alert (overcharge)
00148     if (ina3221_.get_crit_alert(INA3221_CH2)) {
00149         EventEmitter::emit(EventGroup::POWER, PowerEvent::OVERCHARGE);
00150         status_changed = true;
00151     }
00152
00153     // Check power valid alert
00154     if (ina3221_.get_power_valid_alert()) {
00155         EventEmitter::emit(EventGroup::POWER, PowerEvent::POWER_NORMAL);
00156         status_changed = true;
00157     }
00158
00159     recursive_mutex_exit(&powerman_mutex_);
00160     return status_changed;
00161 }
00162 }
```

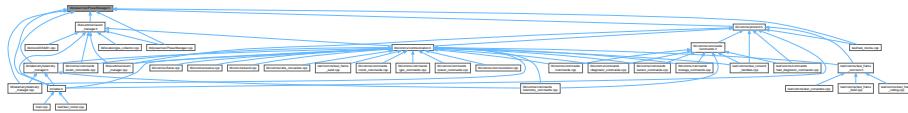
8.65 lib/powerman/PowerManager.h File Reference

```
#include "INA3221/INA3221.h"
#include <map>
#include <string>
#include <hardware/i2c.h>
#include "pico/stl.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](#)

8.66 PowerManager.h

[Go to the documentation of this file.](#)

```

00001 #ifndef POWER_MANAGER_H
00002 #define POWER_MANAGER_H
00003
00004 #include "INA3221/INA3221.h"
00005 #include <map>
00006 #include <string>
00007 #include <hardware/i2c.h>
00008 #include "pico/stdlib.h"
00009 #include "pico/mutex.h"
00010
00011 class PowerManager {
00012 public:
00013     PowerManager(i2c_inst_t* i2c);
00014     bool initialize();
00015     std::string read_device_ids();
00016     float get_current_charge_solar();
00017     float get_current_charge_usb();
00018     float get_current_charge_total();
00019     float get_current_draw();
00020     float get_voltage_battery();
00021     float get_voltage_5v();
00022     void configure(const std::map<std::string, std::string>& config);
00023     bool is_charging_solar();
00024     bool is_charging_usb();
00025     bool check_power_alerts();
00026
00027     static constexpr float SOLAR_CURRENT_THRESHOLD = 50.0f; // mA
00028     static constexpr float USB_CURRENT_THRESHOLD = 50.0f; // mA
00029     static constexpr float VOLTAGE_LOW_THRESHOLD = 4.6f; // V
00030     static constexpr float VOLTAGE_OVERCHARGE_THRESHOLD = 5.3f; // V
00031     static constexpr float FALL_RATE_THRESHOLD = -0.02f; // V/sample
00032     static constexpr int FALLING_TREND_REQUIRED = 3; // samples
00033
00034
  
```

```

00035 private:
00036     INA3221 ina3221_;
00037     bool initialized_;
00038     recursive_mutex_t powerman_mutex_;
00039     bool charging_solar_active_ = false;
00040     bool charging_usb_active_ = false;
00041 };
00042
00043 #endif // POWER_MANAGER_H

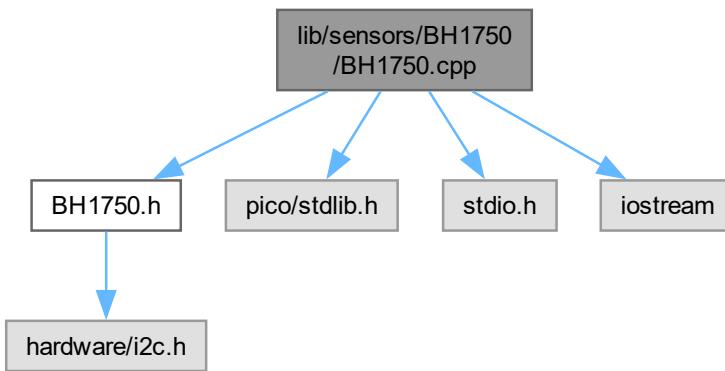
```

8.67 lib/sensors/BH1750/BH1750.cpp File Reference

```

#include "BH1750.h"
#include "pico/stdlib.h"
#include <stdio.h>
#include <iostream>
Include dependency graph for BH1750.cpp:

```



8.68 BH1750.cpp

[Go to the documentation of this file.](#)

```

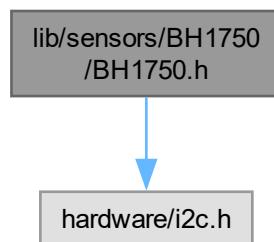
00001 #include "BH1750.h"
00002 #include "pico/stdlib.h"
00003 #include <stdio.h>
00004 #include <iostream>
00005
00006 BH1750::BH1750(uint8_t addr) : _i2c_addr(addr) {}
00007
00008 bool BH1750::begin(Mode mode) {
00009     write8(static_cast<uint8_t>(Mode::POWER_ON));
00010     write8(static_cast<uint8_t>(Mode::RESET));
00011     configure(mode);
00012     configure(BH1750::Mode::POWER_ON);
00013     uint8_t cmd = 0x10; // Continuously H-Resolution Mode
00014     if (i2c_write_blocking(i2c1, _i2c_addr, &cmd, 1, false) == 1) {
00015         std::cout << "BH1750 sensor found at 0x" << std::hex << (int)_i2c_addr << std::endl;
00016         return true;
00017     }
00018     return false;
00019 }
00020
00021 void BH1750::configure(Mode mode) {
00022     uint8_t modeVal = static_cast<uint8_t>(mode);

```

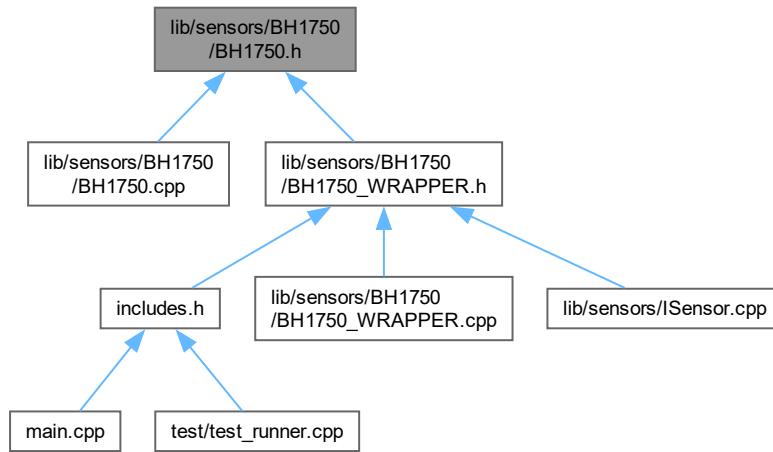
```
00023     switch (mode) {
00024         case Mode::CONTINUOUS_HIGH_RES_MODE:
00025         case Mode::CONTINUOUS_HIGH_RES_MODE_2:
00026         case Mode::CONTINUOUS_LOW_RES_MODE:
00027         case Mode::ONE_TIME_HIGH_RES_MODE:
00028         case Mode::ONE_TIME_HIGH_RES_MODE_2:
00029         case Mode::ONE_TIME_LOW_RES_MODE:
00030             write8(modeVal);
00031             sleep_ms(10);
00032             break;
00033     default:
00034         printf("Invalid measurement mode\n");
00035         break;
00036     }
00037 }
00038
00039 float BH1750::get_light_level() {
00040     uint8_t buffer[2];
00041     i2c_read_blocking(i2c_default, _i2c_addr, buffer, 2, false);
00042     uint16_t level = (buffer[0] << 8) | buffer[1];
00043
00044     float lux = static_cast<float>(level) / 1.2f;
00045     return lux;
00046 }
00047
00048 void BH1750::write8(uint8_t data) {
00049     uint8_t buf[1] = {data};
00050     i2c_write_blocking(i2c_default, _i2c_addr, buf, 1, false);
00051 }
```

8.69 lib/sensors/BH1750/BH1750.h File Reference

```
#include "hardware/i2c.h"
Include dependency graph for BH1750.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- class [BH1750](#)

Macros

- `#define _BH1750_DEVICE_ID 0xE1`
- `#define _BH1750_MTREG_MIN 31`
- `#define _BH1750_MTREG_MAX 254`
- `#define _BH1750_DEFAULT_MTREG 69`

8.69.1 Macro Definition Documentation

8.69.1.1 `_BH1750_DEVICE_ID`

```
#define _BH1750_DEVICE_ID 0xE1
```

Definition at line [7](#) of file [BH1750.h](#).

8.69.1.2 `_BH1750_MTREG_MIN`

```
#define _BH1750_MTREG_MIN 31
```

Definition at line [8](#) of file [BH1750.h](#).

8.69.1.3 _BH1750_MTREG_MAX

```
#define _BH1750_MTREG_MAX 254
```

Definition at line 9 of file [BH1750.h](#).

8.69.1.4 _BH1750_DEFAULT_MTREG

```
#define _BH1750_DEFAULT_MTREG 69
```

Definition at line 10 of file [BH1750.h](#).

8.70 BH1750.h

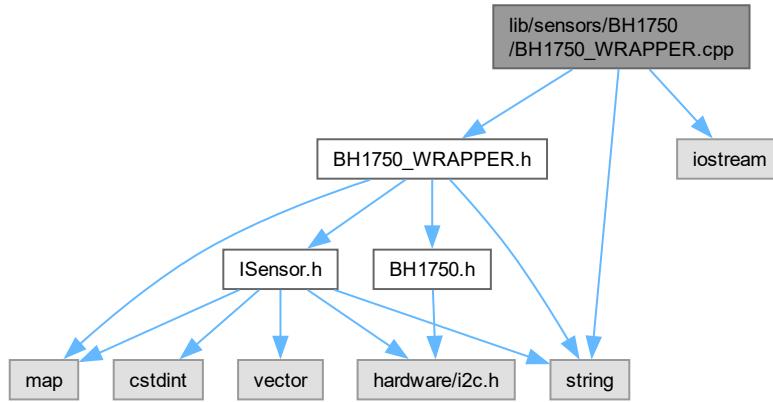
[Go to the documentation of this file.](#)

```
00001 #ifndef __BH1750_H__
00002 #define __BH1750_H__
00003
00004 #include "hardware/i2c.h"
00005
00006 // Define constants
00007 #define _BH1750_DEVICE_ID 0xE1 // Correct content of WHO_AM_I register
00008 #define _BH1750_MTREG_MIN 31
00009 #define _BH1750_MTREG_MAX 254
00010 #define _BH1750_DEFAULT_MTREG 69
00011
00012 class BH1750 {
00013 public:
00014     // Scoped enum for measurement modes
00015     enum class Mode : uint8_t {
00016         UNCONFIGURED_POWER_DOWN = 0x00,
00017         POWER_ON = 0x01,
00018         RESET = 0x07,
00019         CONTINUOUS_HIGH_RES_MODE = 0x10,
00020         CONTINUOUS_HIGH_RES_MODE_2 = 0x11,
00021         CONTINUOUS_LOW_RES_MODE = 0x13,
00022         ONE_TIME_HIGH_RES_MODE = 0x20,
00023         ONE_TIME_HIGH_RES_MODE_2 = 0x21,
00024         ONE_TIME_LOW_RES_MODE = 0x23
00025     };
00026
00027     BH1750(uint8_t addr = 0x23);
00028     bool begin(Mode mode = Mode::CONTINUOUS_HIGH_RES_MODE);
00029     void configure(Mode mode);
00030     float get_light_level();
00031
00032 private:
00033     void write8(uint8_t data);
00034     uint8_t _i2c_addr;
00035 };
00036
00037 #endif // __BH1750_H__
```

8.71 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference

```
#include "BH1750_WRAPPER.h"
#include <string>
```

```
#include <iostream>
Include dependency graph for BH1750_WRAPPER.cpp:
```



8.72 BH1750_WRAPPER.cpp

[Go to the documentation of this file.](#)

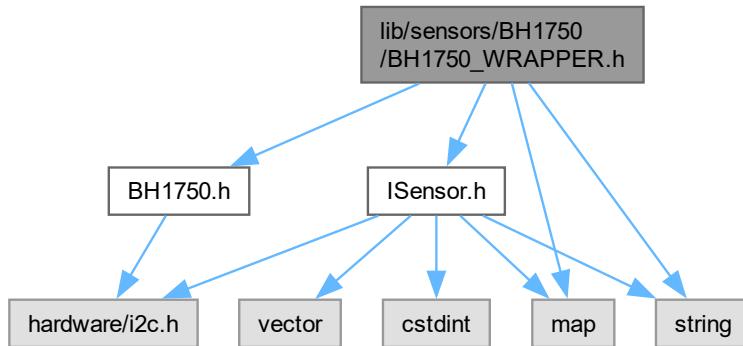
```

00001 // BH1750Wrapper.cpp
00002 #include "BH1750_WRAPPER.h"
00003 #include <string>
00004 #include <iostream>
00005
00006 BH1750Wrapper::BH1750Wrapper() {
00007     sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
00008 }
00009
00010 bool BH1750Wrapper::init() {
00011     initialized_ = sensor_.begin();
00012     return initialized_;
00013 }
00014
00015 float BH1750Wrapper::read_data(SensorDataTypeIdentifier type) {
00016     if (type == SensorDataTypeIdentifier::LIGHT_LEVEL) {
00017         return sensor_.get_light_level();
00018     }
00019     return 0.0f;
00020 }
00021
00022 bool BH1750Wrapper::is_initialized() const {
00023     return initialized_;
00024 }
00025
00026 SensorType BH1750Wrapper::get_type() const {
00027     return SensorType::LIGHT;
00028 }
00029
00030 bool BH1750Wrapper::configure(const std::map<std::string, std::string>& config) {
00031     for (const auto& [key, value] : config) {
00032         if (key == "measurement_mode") {
00033             if (value == "continuously_high_resolution") {
00034                 sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
00035             }
00036             else if (value == "continuously_high_resolution_2") {
00037                 sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE_2);
00038             }
00039             else if (value == "continuously_low_resolution") {
00040                 sensor_.configure(BH1750::Mode::CONTINUOUS_LOW_RES_MODE);
00041             }
00042             else if (value == "one_time_high_resolution") {
00043                 sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE);
00044             }
00045             else if (value == "one_time_high_resolution_2") {
  
```

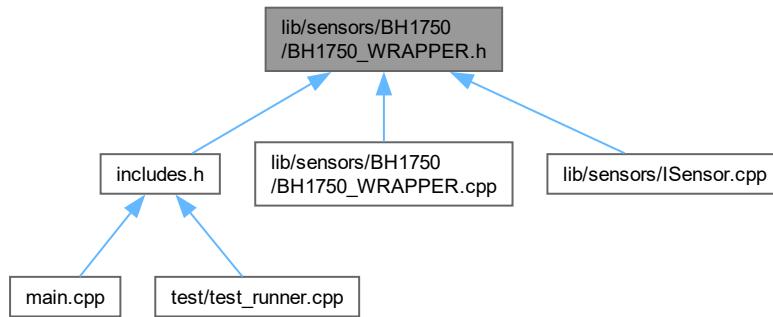
```
00046     sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE_2);
00047 }
00048 else if (value == "one_time_low_resolution") {
00049     sensor_.configure(BH1750::Mode::ONE_TIME_LOW_RES_MODE);
00050 }
00051 else {
00052     std::cerr << "[BH1750Wrapper] Unknown measurement_mode value: " << value << std::endl;
00053     return false;
00054 }
00055 // Handle additional configuration keys here
00056 else {
00057     std::cerr << "[BH1750Wrapper] Unknown configuration key: " << key << std::endl;
00058     return false;
00059 }
00060 }
00061 }
00062 return true;
00063 }
```

8.73 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](#)

8.74 BH1750_WRAPPER.h

[Go to the documentation of this file.](#)

```

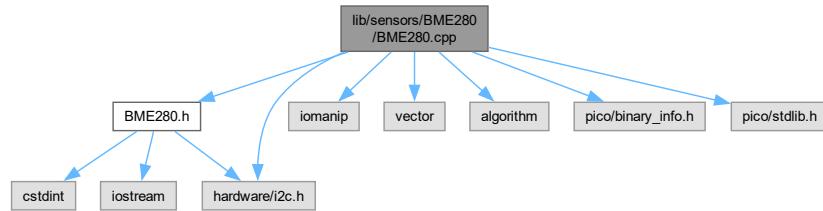
00001 #ifndef BH1750_WRAPPER_H
00002 #define BH1750_WRAPPER_H
00003
00004 #include "ISensor.h"
00005 #include "BH1750.h"
00006 #include <map>
00007 #include <string>
00008
00009 class BH1750Wrapper : public ISensor {
00010 private:
00011     BH1750 sensor_;
00012     bool initialized_ = false;
00013
00014 public:
00015     BH1750Wrapper();
00016     int get_i2c_addr();
00017     bool init() override;
00018     float read_data(SensorDataTypeIdentifier type) override;
00019     bool is_initialized() const override;
00020     SensorType get_type() const override;
00021
00022     bool configure(const std::map<std::string, std::string>& config);
00023
00024     uint8_t get_address() const override {
00025         return 0x23;
00026     }
00027 };
00028
00029 #endif // BH1750_WRAPPER_H
  
```

8.75 lib/sensors/BME280/BME280.cpp File Reference

```

#include "BME280.h"
#include <iomanip>
#include <vector>
  
```

```
#include <algorithm>
#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdc.h"
Include dependency graph for BME280.cpp:
```



8.76 BME280.cpp

[Go to the documentation of this file.](#)

```

00001 // BME280.cpp
00002
00003 #include "BME280.h"
00004
00005 #include <iomanip>
00006 #include <vector>
00007 #include <algorithm>
00008 #include "hardware/i2c.h"
00009 #include "pico/binary_info.h"
00010 #include "pico/stdc.h"
00011
00012 // BME280 (BME280) Class Implementation
00013
00014 BME280::BME280(i2c_inst_t* i2cPort, uint8_t address)
00015     : i2c_port(i2cPort), device_addr(address), calib_params{}, initialized_(false), t_fine(0) {
00016 }
00017
00018 bool BME280::init() {
00019     if (!i2c_port) {
00020         std::cerr << "Invalid I2C port.\n";
00021         return false;
00022     }
00023
00024     // Check device ID to confirm it's a BME280
00025     uint8_t reg = 0xD0; // Chip ID register
00026     uint8_t chip_id = 0;
00027     int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
00028     if (ret != 1) {
00029         std::cerr << "Failed to write to BME280.\n";
00030         return false;
00031     }
00032     ret = i2c_read_blocking(i2c_port, device_addr, &chip_id, 1, false);
00033     if (ret != 1) {
00034         std::cerr << "Failed to read chip ID from BME280.\n";
00035         return false;
00036     }
00037     if (chip_id != 0x60) {
00038         std::cerr << "Device is not a BME280.\n";
00039         return false;
00040     }
00041
00042     // Configure sensor
00043     if (!configure_sensor()) {
00044         std::cerr << "Failed to configure BME280 sensor.\n";
00045         return false;
00046     }
00047
00048     // Retrieve calibration parameters
00049     if (!get_calibration_parameters()) {
00050         std::cerr << "Failed to retrieve calibration parameters.\n";
00051         return false;
00052     }

```

```

00053     initialized_ = true;
00054     std::cout << "BME280 sensor initialized_ successfully.\n";
00055     return true;
00056 }
00058
00059 void BME280::reset() {
00060     uint8_t buf[2] = { REG_RESET, 0xB6 };
00061     int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
00062     if (ret != 2) {
00063         std::cerr << "Failed to reset BME280 sensor.\n";
00064     }
00065     sleep_ms(10); // Wait for reset to complete
00066 }
00067
00068 bool BME280::read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity) {
00069     if (!initialized_) {
00070         std::cerr << "BME280 not initialized_.\n";
00071         return false;
00072     }
00073
00074     // Define the starting register address
00075     uint8_t start_reg = REG_PRESSURE_MSB;
00076     // Total bytes to read: 3 (pressure) + 3 (temperature) + 2 (humidity) = 8
00077     uint8_t buf[8] = {0};
00078
00079     // Write the starting register address
00080     int ret = i2c_write_blocking(i2c_port, device_addr, &start_reg, 1, true);
00081     if (ret != 1) {
00082         std::cerr << "Failed to write starting register address to BME280.\n";
00083         return false;
00084     }
00085
00086     // Read data
00087     ret = i2c_read_blocking(i2c_port, device_addr, buf, 8, false);
00088     if (ret != 8) {
00089         std::cerr << "Failed to read data from BME280.\n";
00090         return false;
00091     }
00092
00093     // Combine bytes to form raw values
00094     *pressure = ((int32_t)buf[0] << 12) | ((int32_t)buf[1] << 4) | ((int32_t)(buf[2] >> 4));
00095     *temperature = ((int32_t)buf[3] << 12) | ((int32_t)buf[4] << 4) | ((int32_t)(buf[5] >> 4));
00096     *humidity = ((int32_t)buf[6] << 8) | (int32_t)buf[7];
00097
00098     return true;
00099 }
00100
00101 float BME280::convert_temperature(int32_t temp_raw) const {
00102     int32_t var1, var2;
00103     var1 = (((temp_raw >> 3) - ((int32_t)calib_params.dig_t1 << 1))) * ((int32_t)calib_params.dig_t2)
00104     >> 11;
00105     var2 = (((((temp_raw >> 4) - ((int32_t)calib_params.dig_t1)) * ((temp_raw >> 4) -
00106     ((int32_t)calib_params.dig_t1)) >> 12) * ((int32_t)calib_params.dig_t3)) >> 14;
00107     t_fine = var1 + var2;
00108     float T = (t_fine * 5 + 128) >> 8;
00109     return T / 100.0f;
00110 }
00111
00112 float BME280::convert_pressure(int32_t pressure_raw) const {
00113     int64_t var1, var2, p;
00114     var1 = ((int64_t)t_fine) - 128000;
00115     var2 = var1 * var1 * (int64_t)calib_params.dig_p6;
00116     var2 = var2 + ((var1 * (int64_t)calib_params.dig_p5) << 17);
00117     var2 = var2 + (((int64_t)calib_params.dig_p4) << 35);
00118     var1 = ((var1 * var1 * (int64_t)calib_params.dig_p3) >> 8) + ((var1 * (int64_t)calib_params.dig_p2)
00119     << 12);
00120     var1 = (((int64_t)1 << 47) + var1) * ((int64_t)calib_params.dig_p1) >> 33;
00121
00122     if (var1 == 0) {
00123         return 0.0f; // avoid exception caused by division by zero
00124     }
00125     p = 1048576 - pressure_raw;
00126     p = ((p << 31) - var2) * 3125) / var1;
00127     var1 = (((int64_t)calib_params.dig_p9) * (p >> 13) * (p >> 13)) >> 25;
00128     var2 = (((int64_t)calib_params.dig_p8) * p) >> 19;
00129
00130     p = ((p + var1 + var2) >> 8) + (((int64_t)calib_params.dig_p7) << 4);
00131     return (float)p / 25600.0f; // in hPa
00132 }
00133
00134 float BME280::convert_humidity(int32_t humidity_raw) const {
00135     int32_t v_x1_u32r;
00136     v_x1_u32r = t_fine - 76800;
00137     v_x1_u32r = (((humidity_raw << 14) - ((int32_t)calib_params.dig_h4 << 20) -
00138     ((int32_t)calib_params.dig_h5 * v_x1_u32r)) + 16384) >> 15) *
00139     (((((v_x1_u32r * (int32_t)calib_params.dig_h6) >> 10) * (((v_x1_u32r *

```

```

00136     (int32_t)calib_params.dig_h3) >> 11) + 32768)) >> 10) + 2097152) *
00137     (int32_t)calib_params.dig_h2 + 8192) >> 14));
00138 v_x1_u32r = std::max(v_x1_u32r, (int32_t)0);
00139 v_x1_u32r = std::min(v_x1_u32r, (int32_t)419430400);
00140 float h = v_x1_u32r >> 12;
00141 return h / 1024.0f;
00142
00143 bool BME280::get_calibration_parameters() {
00144     // Read temperature and pressure calibration data (0x88 to 0xA1)
00145     uint8_t calib_data[26];
00146     uint8_t reg = REG_DIG_T1_LSB;
00147     int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
00148     if (ret != 1) {
00149         std::cerr << "Failed to write to BME280.\n";
00150         return false;
00151     }
00152     ret = i2c_read_blocking(i2c_port, device_addr, calib_data, 26, false);
00153     if (ret != 26) {
00154         std::cerr << "Failed to read calibration data from BME280.\n";
00155         return false;
00156     }
00157
00158     // Parse temperature calibration data
00159     calib_params.dig_t1 = (uint16_t)(calib_data[1] << 8 | calib_data[0]);
00160     calib_params.dig_t2 = (int16_t)(calib_data[3] << 8 | calib_data[2]);
00161     calib_params.dig_t3 = (int16_t)(calib_data[5] << 8 | calib_data[4]);
00162
00163     // Parse pressure calibration data
00164     calib_params.dig_p1 = (uint16_t)(calib_data[7] << 8 | calib_data[6]);
00165     calib_params.dig_p2 = (int16_t)(calib_data[9] << 8 | calib_data[8]);
00166     calib_params.dig_p3 = (int16_t)(calib_data[11] << 8 | calib_data[10]);
00167     calib_params.dig_p4 = (int16_t)(calib_data[13] << 8 | calib_data[12]);
00168     calib_params.dig_p5 = (int16_t)(calib_data[15] << 8 | calib_data[14]);
00169     calib_params.dig_p6 = (int16_t)(calib_data[17] << 8 | calib_data[16]);
00170     calib_params.dig_p7 = (int16_t)(calib_data[19] << 8 | calib_data[18]);
00171     calib_params.dig_p8 = (int16_t)(calib_data[21] << 8 | calib_data[20]);
00172     calib_params.dig_p9 = (int16_t)(calib_data[23] << 8 | calib_data[22]);
00173
00174     calib_params.dig_h1 = calib_data[25];
00175
00176     // Read humidity calibration data (0xE1 to 0xE7)
00177     reg = 0xE1;
00178     ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
00179     if (ret != 1) {
00180         std::cerr << "Failed to write to BME280 for humidity calibration.\n";
00181         return false;
00182     }
00183
00184     uint8_t hum_calib_data[7];
00185     ret = i2c_read_blocking(i2c_port, device_addr, hum_calib_data, 7, false);
00186     if (ret != 7) {
00187         std::cerr << "Failed to read humidity calibration data from BME280.\n";
00188         return false;
00189     }
00190
00191     // Parse humidity calibration data
00192     calib_params.dig_h2 = (int16_t)(hum_calib_data[1] << 8 | hum_calib_data[0]);
00193     calib_params.dig_h3 = hum_calib_data[2];
00194     calib_params.dig_h4 = (int16_t)((hum_calib_data[3] << 4) | (hum_calib_data[4] & 0x0F));
00195     calib_params.dig_h5 = (int16_t)((hum_calib_data[5] << 4) | (hum_calib_data[4] >> 4));
00196     calib_params.dig_h6 = (int8_t)hum_calib_data[6];
00197
00198     return true;
00199 }
00200
00201 bool BME280::configure_sensor() {
00202     uint8_t buf[2];
00203
00204     // Set humidity oversampling (must be set before ctrl_meas)
00205     buf[0] = REG_CTRL_HUM;
00206     buf[1] = 0x05; // Humidity oversampling x16
00207     int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
00208     if (ret != 2) {
00209         std::cerr << "Failed to write CTRL_HUM to BME280.\n";
00210         return false;
00211     }
00212
00213     // Write config register
00214     buf[0] = REG_CONFIG;
00215     buf[1] = 0x00; // Default settings
00216     ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
00217     if (ret != 2) {
00218         std::cerr << "Failed to write CONFIG to BME280.\n";
00219         return false;
00220     }
00221

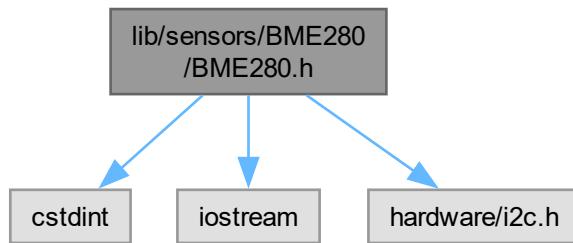
```

```

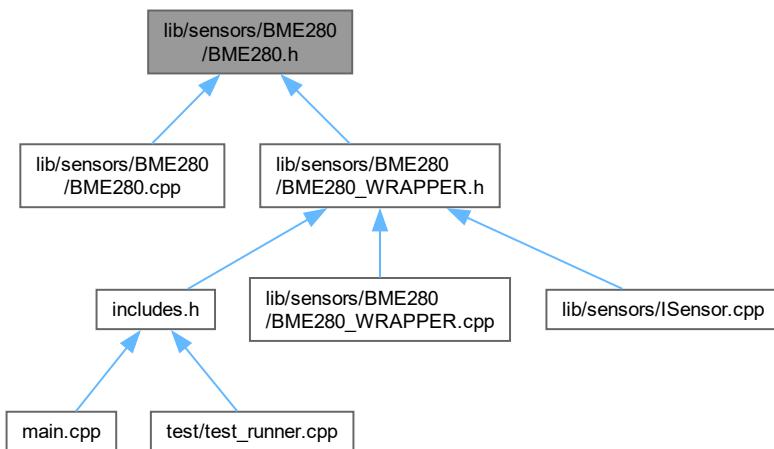
00222 // Write ctrl_meas register
00223 buf[0] = REG_CTRL_MEAS;
00224 buf[1] = 0xB7; // Temp and pressure oversampling x16, normal mode
00225 ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
00226 if (ret !=2) {
00227     std::cerr << "Failed to write CTRL_MEAS to BME280.\n";
00228     return false;
00229 }
00230
00231 return true;
00232 }
```

8.77 lib/sensors/BME280/BME280.h File Reference

```
#include <cstdint>
#include <iostream>
#include "hardware/i2c.h"
Include dependency graph for BME280.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- struct **BME280CalibParam**
- class **BME280**

8.78 BME280.h

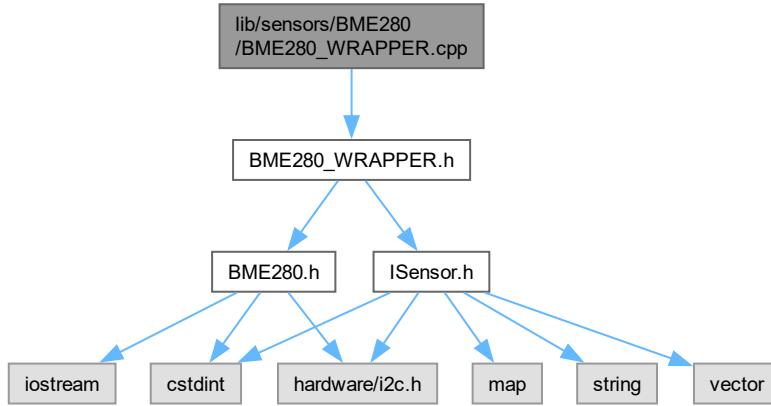
Go to the documentation of this file.

```
00001 // BME280.h
00002
00003 #ifndef BME280_H
00004 #define BME280_H
00005
00006 #include <cstdint>
00007 #include <iostream>
00008 #include "hardware/i2c.h"
00009
00010 // Calibration parameters structure
00011 struct BME280CalibParam {
00012     // Temperature parameters
00013     uint16_t dig_t1;
00014     int16_t dig_t2;
00015     int16_t dig_t3;
00016
00017     // Pressure parameters
00018     uint16_t dig_p1;
00019     int16_t dig_p2;
00020     int16_t dig_p3;
00021     int16_t dig_p4;
00022     int16_t dig_p5;
00023     int16_t dig_p6;
00024     int16_t dig_p7;
00025     int16_t dig_p8;
00026     int16_t dig_p9;
00027
00028     // Humidity parameters
00029     uint8_t dig_h1;
00030     int16_t dig_h2;
00031     uint8_t dig_h3;
00032     int16_t dig_h4;
00033     int16_t dig_h5;
00034     int8_t dig_h6;
00035 };
00036
00037 // BME280 Class Definition
00038 class BME280 {
00039 public:
00040     // I2C Address Options
00041     static constexpr uint8_t ADDR_SDO_LOW = 0x76;
00042     static constexpr uint8_t ADDR_SDO_HIGH = 0x77;
00043
00044     // Constructor
00045     BME280(i2c_inst_t* i2cPort, uint8_t address = ADDR_SDO_LOW);
00046
00047     // Initialize the sensor
00048     bool init();
00049
00050     // Reset the sensor
00051     void reset();
00052
00053     // Read all raw data: temperature, pressure, and humidity
00054     bool read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity);
00055
00056     // Convert raw data to actual values
00057     float convert_temperature(int32_t temp_raw) const;
00058     float convert_pressure(int32_t pressure_raw) const;
00059     float convert_humidity(int32_t humidity_raw) const;
00060
00061 private:
00062     // Configure the sensor
00063     bool configure_sensor();
00064
00065     // Retrieve calibration parameters from the sensor
00066     bool get_calibration_parameters();
00067
00068     // I2C port and device address
00069     i2c_inst_t* i2c_port;
00070     uint8_t device_addr;
00071
```

```
00072 // Calibration parameters
00073 BME280CalibParam calib_params;
00074
00075 // Initialization status
00076 bool initialized_;
00077
00078 // Fine temperature parameter needed for compensation
00079 mutable int32_t t_fine;
00080
00081 // Register Definitions
00082 static constexpr uint8_t REG_CONFIG = 0xF5;
00083 static constexpr uint8_t REG_CTRL_MEAS = 0xF4;
00084 static constexpr uint8_t REG_CTRL_HUM = 0xF2;
00085 static constexpr uint8_t REG_RESET = 0xE0;
00086
00087 static constexpr uint8_t REG_PRESSURE_MSB = 0xF7;
00088 static constexpr uint8_t REG_TEMPERATURE_MSB = 0xFA;
00089 static constexpr uint8_t REG_HUMIDITY_MSB = 0xFD;
00090
00091 // Calibration Registers
00092 static constexpr uint8_t REG_DIG_T1_LSB = 0x88;
00093 static constexpr uint8_t REG_DIG_T1_MSB = 0x89;
00094 static constexpr uint8_t REG_DIG_T2_LSB = 0x8A;
00095 static constexpr uint8_t REG_DIG_T2_MSB = 0x8B;
00096 static constexpr uint8_t REG_DIG_T3_LSB = 0x8C;
00097 static constexpr uint8_t REG_DIG_T3_MSB = 0x8D;
00098
00099 static constexpr uint8_t REG_DIG_P1_LSB = 0x8E;
00100 static constexpr uint8_t REG_DIG_P1_MSB = 0x8F;
00101 static constexpr uint8_t REG_DIG_P2_LSB = 0x90;
00102 static constexpr uint8_t REG_DIG_P2_MSB = 0x91;
00103 static constexpr uint8_t REG_DIG_P3_LSB = 0x92;
00104 static constexpr uint8_t REG_DIG_P3_MSB = 0x93;
00105 static constexpr uint8_t REG_DIG_P4_LSB = 0x94;
00106 static constexpr uint8_t REG_DIG_P4_MSB = 0x95;
00107 static constexpr uint8_t REG_DIG_P5_LSB = 0x96;
00108 static constexpr uint8_t REG_DIG_P5_MSB = 0x97;
00109 static constexpr uint8_t REG_DIG_P6_LSB = 0x98;
00110 static constexpr uint8_t REG_DIG_P6_MSB = 0x99;
00111 static constexpr uint8_t REG_DIG_P7_LSB = 0x9A;
00112 static constexpr uint8_t REG_DIG_P7_MSB = 0x9B;
00113 static constexpr uint8_t REG_DIG_P8_LSB = 0x9C;
00114 static constexpr uint8_t REG_DIG_P8_MSB = 0x9D;
00115 static constexpr uint8_t REG_DIG_P9_LSB = 0x9E;
00116 static constexpr uint8_t REG_DIG_P9_MSB = 0x9F;
00117
00118 // Humidity Calibration Registers
00119 static constexpr uint8_t REG_DIG_H1 = 0xA1;
00120 static constexpr uint8_t REG_DIG_H2 = 0xE1;
00121 static constexpr uint8_t REG_DIG_H3 = 0xE3;
00122 static constexpr uint8_t REG_DIG_H4 = 0xE4;
00123 static constexpr uint8_t REG_DIG_H5 = 0xE5;
00124 static constexpr uint8_t REG_DIG_H6 = 0xE7;
00125
00126 // Number of calibration parameters to read
00127 static constexpr size_t NUM_CALIB_PARAMS = 24;
00128 };
00129
00130 #endif // BME280_H
```

8.79 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference

```
#include "BME280_WRAPPER.h"
Include dependency graph for BME280_WRAPPER.cpp:
```



8.80 BME280_WRAPPER.cpp

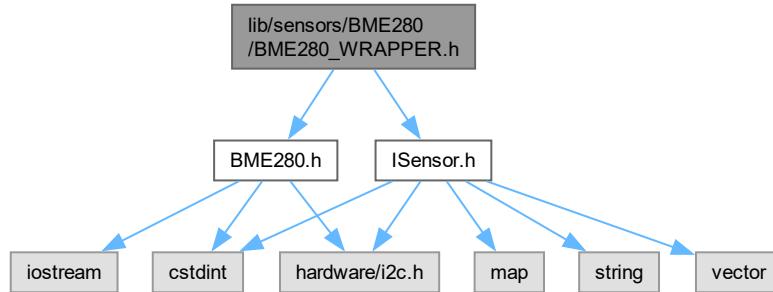
[Go to the documentation of this file.](#)

```

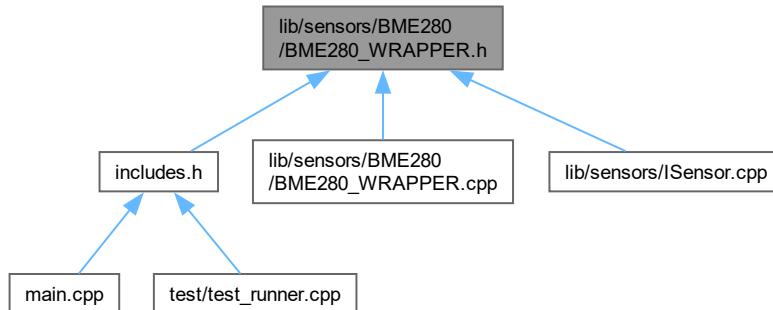
00001 #include "BME280_WRAPPER.h"
00002
00003 BME280Wrapper::BME280Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {}
00004
00005 bool BME280Wrapper::init() {
00006     initialized_ = sensor_.init();
00007     return initialized_;
00008 }
00009
00010 float BME280Wrapper::read_data(SensorDataTypeIdentifier type) {
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(const std::map<std::string, std::string>& config) {
00035     return true;
00036 }
```

8.81 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](#)

8.82 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
```

```

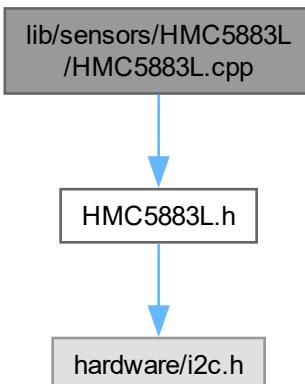
00008 class BME280Wrapper : public ISensor {
00009     private:
0010         BME280 sensor_;
0011         bool initialized_ = false;
0012
0013     public:
0014         BME280Wrapper(i2c_inst_t* i2c);
0015
0016         bool init() override;
0017         float read_data(SensorDataTypeIdentifier type) override;
0018         bool is_initialized() const override;
0019         SensorType get_type() const override;
0020         bool configure(const std::map<std::string, std::string>& config) override;
0021
0022         uint8_t get_address() const override {
0023             return 0x76;
0024         }
0025
0026     };
0027
0028 #endif // BME280_WRAPPER_H

```

8.83 lib/sensors/HMC5883L/HMC5883L.cpp File Reference

#include "HMC5883L.h"

Include dependency graph for HMC5883L.cpp:



8.84 HMC5883L.cpp

[Go to the documentation of this file.](#)

```

00001 #include "HMC5883L.h"
00002
00003 HMC5883L::HMC5883L(i2c_inst_t* i2c, uint8_t address) : i2c(i2c), address(address) {}
00004
00005 bool HMC5883L::init() {
00006     // Continuous measurement mode, 15Hz data output rate
00007     if (!write_register(0x00, 0x70)) return false;
00008     if (!write_register(0x01, 0x20)) return false;
00009     if (!write_register(0x02, 0x00)) return false;
00010     return true;
00011 }
00012
00013 bool HMC5883L::read(int16_t& x, int16_t& y, int16_t& z) {

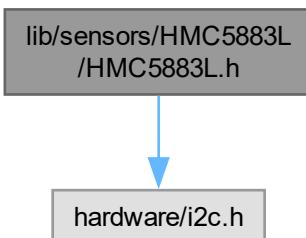
```

```
00014     uint8_t buffer[6];
00015     if (!read_register(0x03, buffer, 6)) return false;
00016
00017     x = (buffer[0] << 8) | buffer[1];
00018     z = (buffer[2] << 8) | buffer[3];
00019     y = (buffer[4] << 8) | buffer[5];
00020
00021     if (x > 32767) x -= 65536;
00022     if (y > 32767) y -= 65536;
00023     if (z > 32767) z -= 65536;
00024
00025     return true;
00026 }
00027
00028 bool HMC5883L::write_register(uint8_t reg, uint8_t value) {
00029     uint8_t buffer[2] = {reg, value};
00030     return i2c_write_blocking(i2c, address, buffer, 2, false) == 2;
00031 }
00032
00033 bool HMC5883L::read_register(uint8_t reg, uint8_t* buffer, size_t length) {
00034     if (i2c_write_blocking(i2c, address, &reg, 1, true) != 1) return false;
00035     return i2c_read_blocking(i2c, address, buffer, length, false) == length;
00036 }
```

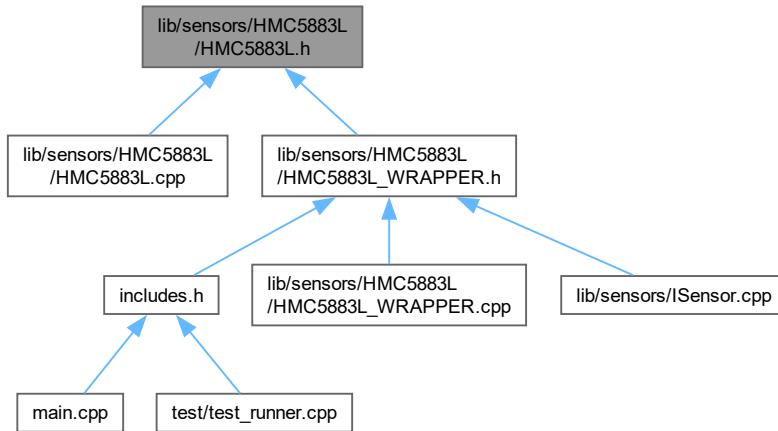
8.85 lib/sensors/HMC5883L/HMC5883L.h File Reference

#include "hardware/i2c.h"

Include dependency graph for HMC5883L.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [HMC5883L](#)

8.86 HMC5883L.h

[Go to the documentation of this file.](#)

```

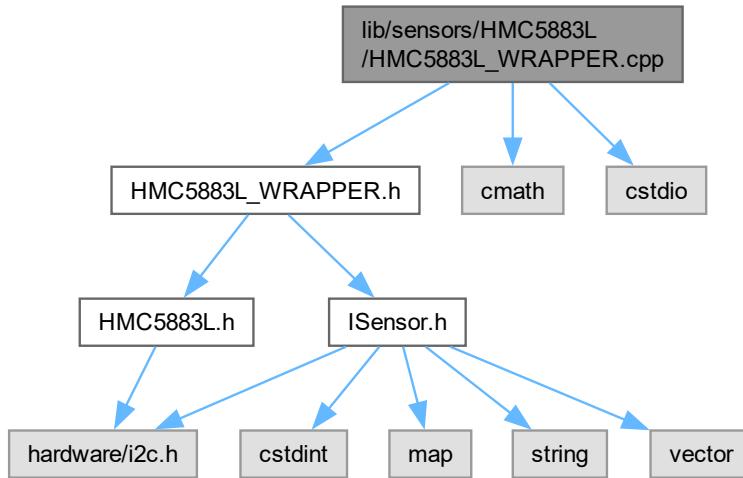
00001 #ifndef HMC5883L_H
00002 #define HMC5883L_H
00003
00004 #include "hardware/i2c.h"
00005
00006 class HMC5883L {
00007 public:
00008     HMC5883L(i2c_inst_t* i2c, uint8_t address = 0x0D);
00009     bool init();
00010     bool read(int16_t& x, int16_t& y, int16_t& z);
00011
00012 private:
00013     i2c_inst_t* i2c;
00014     uint8_t address;
00015
00016     bool write_register(uint8_t reg, uint8_t value);
00017     bool read_register(uint8_t reg, uint8_t* buffer, size_t length);
00018 };
00019
00020 #endif
  
```

8.87 lib/sensors/HMC5883L/HMC5883L_WRAPPER.cpp File Reference

```

#include "HMC5883L_WRAPPER.h"
#include <cmath>
  
```

```
#include <cstdio>
Include dependency graph for HMC5883L_WRAPPER.cpp:
```



8.88 HMC5883L_WRAPPER.cpp

[Go to the documentation of this file.](#)

```

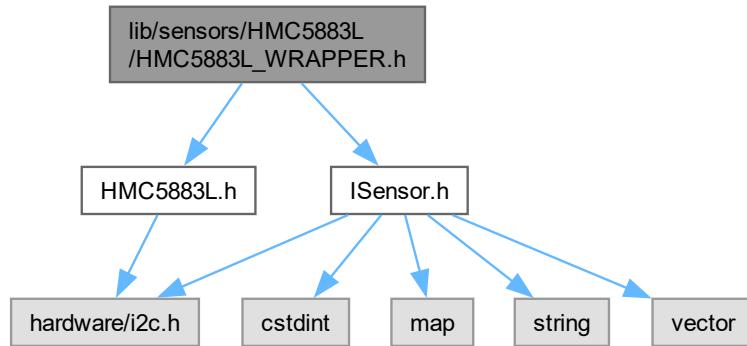
00001 #include "HMC5883L_WRAPPER.h"
00002 #include <cmath>
00003 #include <cstdio>
00004
00005 HMC5883LWrapper::HMC5883LWrapper(i2c_inst_t* i2c) : sensor_(i2c), initialized_(false) {}
00006
00007 bool HMC5883LWrapper::init() {
00008     initialized_ = sensor_.init();
00009     return initialized_;
00010 }
00011
00012 float HMC5883LWrapper::read_data(SensorDataTypeIdentifier type) {
00013     if (!initialized_) return 0.0f;
00014
00015     int16_t x, y, z;
00016     if (!sensor_.read(x, y, z)) return 0.0f;
00017
00018     const float LSB_TO_UT = 100.0 / 1090.0;
00019     float x_uT = x * LSB_TO_UT;
00020     float y_uT = y * LSB_TO_UT;
00021     float z_uT = z * LSB_TO_UT;
00022
00023     switch (type) {
00024         case SensorDataTypeIdentifier::MAG_FIELD_X:
00025             return x_uT;
00026         case SensorDataTypeIdentifier::MAG_FIELD_Y:
00027             return y_uT;
00028         case SensorDataTypeIdentifier::MAG_FIELD_Z:
00029             return z_uT;
00030         default:
00031             return 0.0f;
00032     }
00033 }
00034
00035 bool HMC5883LWrapper::is_initialized() const {
00036     return initialized_;
00037 }
00038
00039 SensorType HMC5883LWrapper::get_type() const {
00040     return SensorType::MAGNETOMETER;
  
```

```

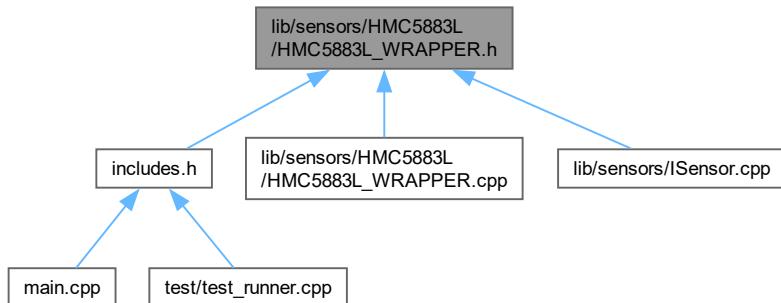
00041 }
00042
00043 bool HMC5883LWrapper::configure(const std::map<std::string, std::string>& config) {
00044     // Configuration logic if needed
00045     return true;
00046 }
```

8.89 lib/sensors/HMC5883L/HMC5883L_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "HMC5883L.h"
Include dependency graph for HMC5883L_WRAPPER.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- class [HMC5883LWrapper](#)

8.90 HMC5883L_WRAPPER.h

[Go to the documentation of this file.](#)

```

00001 #ifndef HMC5883L_WRAPPER_H
00002 #define HMC5883L_WRAPPER_H
00003
00004 #include "ISensor.h"
00005 #include "HMC5883L.h"
00006
00007 class HMC5883LWrapper : public ISensor {
00008 public:
00009     HMC5883LWrapper(i2c_inst_t* i2c);
00010     bool init() override;
00011     float read_data(SensorDataTypeIdentifier type) override;
00012     bool is_initialized() const override;
00013     SensorType get_type() const override;
00014     bool configure(const std::map<std::string, std::string>& config) override;
00015
00016     uint8_t get_address() const override {
00017         return 0x0D;
00018     }
00019
00020 private:
00021     HMC5883L sensor_;
00022     bool initialized_;
00023 };
00024
00025 #endif

```

8.91 lib/sensors/ISensor.cpp File Reference

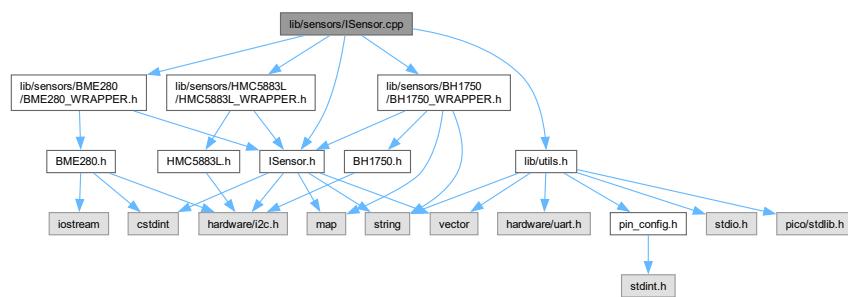
Implements the [SensorWrapper](#) class for managing different sensor types.

```

#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/sensors/HMC5883L/HMC5883L_WRAPPER.h"
#include "lib/utils.h"

Include dependency graph for ISensor.cpp:

```



8.91.1 Detailed Description

Implements the [SensorWrapper](#) class for managing different sensor types.

This file provides the implementation for initializing, configuring, and reading data from various sensors.

Definition in file [ISensor.cpp](#).

8.92 ISensor.cpp

[Go to the documentation of this file.](#)

```

00001 // ISensor.cpp
00002 #include "ISensor.h"
00003 #include "lib/sensors/BH1750/BH1750_WRAPPER.h"
00004 #include "lib/sensors/BME280/BME280_WRAPPER.h"
00005 #include "lib/sensors/HMC5883L/HMC5883L_WRAPPER.h"
00006 #include "lib/utils.h"
00007
00014
00019
00024 SensorWrapper& SensorWrapper::get_instance() {
00025     static SensorWrapper instance;
00026     return instance;
00027 }
00028
00029
00033 SensorWrapper::SensorWrapper() = default;
00034
00035
00042 bool SensorWrapper::sensor_init(SensorType type, i2c_inst_t* i2c) {
00043     switch(type) {
00044         case SensorType::LIGHT:
00045             sensors[type] = new BH1750Wrapper();
00046             break;
00047         case SensorType::ENVIRONMENT:
00048             sensors[type] = new BME280Wrapper(i2c);
00049             break;
00050         case SensorType::IMU:
00051             //sensors[type] = new MPU6050Wrapper(i2c);
00052             break;
00053         case SensorType::MAGNETOMETER:
00054             sensors[type] = new HMC5883LWrapper(i2c);
00055             break;
00056     }
00057     return sensors[type]->init();
00058 }
00059
00060
00067 bool SensorWrapper::sensor_configure(SensorType type, const std::map<std::string, std::string>& config) {
00068     auto it = sensors.find(type);
00069     if (it != sensors.end() && it->second->is_initialized()) {
00070         return it->second->configure(config);
00071     }
00072     std::cerr << "Sensor not initialized or not found: " << static_cast<int>(type) << std::endl;
00073     return false;
00074 }
00075
00076
00083 float SensorWrapper::sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType) {
00084     auto it = sensors.find(sensorType);
00085     if (it != sensors.end() && it->second->is_initialized()) {
00086         return it->second->read_data(dataType);
00087     }
00088     return 0.0f;
00089 }
00090
00091
00096 std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::get_available_sensors() {
00097     std::vector<std::pair<SensorType, uint8_t>> available_sensors;
00098
00099     for (const auto& sensor_pair : sensors) {
00100         if (sensor_pair.second->is_initialized()) {
00101             available_sensors.push_back({sensor_pair.first, sensor_pair.second->get_address()});
00102         }
00103     }
00104
00105     return available_sensors;
00106 }
00107
00108
00114 std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::scan_connected_sensors(i2c_inst_t* i2c) {
00115     std::vector<std::pair<SensorType, uint8_t>> connected_sensors;
00116
00117     // Define the address ranges to check for each sensor type
00118     struct SensorAddressInfo {
00119         SensorType type;
00120         std::vector<uint8_t> addresses;
00121     };
00122
00123     std::vector<SensorAddressInfo> sensor_addresses = {
00124         {SensorType::LIGHT, {0x23, 0x5C}},           // BH1750 addresses
00125         {SensorType::ENVIRONMENT, {0x76, 0x77}},      // BME280 addresses

```

```

00126     {SensorType::MAGNETOMETER, {0x0D, 0x1E}},           // HMC5883L addresses
00127     {SensorType::IMU, {0x68, 0x69}}                    // MPU6050 addresses
00128 };
00129
00130 // Buffer for receiving ACK/NACK
00131 uint8_t rxdata;
00132
00133 for (const auto& sensor_info : sensor_addresses) {
00134     for (uint8_t addr : sensor_info.addresses) {
00135         // Try to read a byte from the device to see if it ACKs
00136         int result = i2c_read_blocking(i2c, addr, &rxdata, 1, false);
00137         if (result >= 0) {
00138             // We received an ACK, so the device exists
00139             connected_sensors.push_back({sensor_info.type, addr});
00140             uart_print("Found sensor at address 0x" + std::to_string(addr),
00141             VerbosityLevel::DEBUG);
00142         }
00143     }
00144 }
00145 return connected_sensors;
00146 }
00147

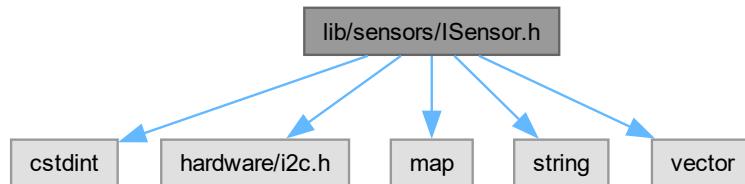
```

8.93 lib/sensors/ISensor.h File Reference

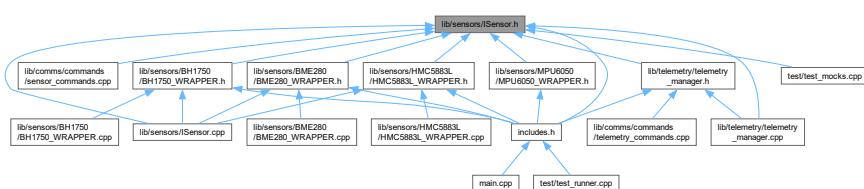
```

#include <cstdint>
#include "hardware/i2c.h"
#include <map>
#include <string>
#include <vector>
Include dependency graph for ISensor.h:

```



This graph shows which files directly or indirectly include this file:



Classes

- class [ISensor](#)
- class [SensorWrapper](#)

Manages different sensor types and provides a unified interface for accessing sensor data.

Enumerations

- enum class `SensorType` { `LIGHT` , `ENVIRONMENT` , `MAGNETOMETER` , `IMU` }
- enum class `SensorDataTypelIdentifier` {
`LIGHT_LEVEL` , `TEMPERATURE` , `PRESSURE` , `HUMIDITY` ,
`MAG_FIELD_X` , `MAG_FIELD_Y` , `MAG_FIELD_Z` , `GYRO_X` ,
`GYRO_Y` , `GYRO_Z` , `ACCEL_X` , `ACCEL_Y` ,
`ACCEL_Z` }

8.93.1 Enumeration Type Documentation

8.93.1.1 SensorType

```
enum class SensorType [strong]
```

Enumerator

<code>LIGHT</code>	
<code>ENVIRONMENT</code>	
<code>MAGNETOMETER</code>	
<code>IMU</code>	

Definition at line 11 of file `ISensor.h`.

8.93.1.2 SensorDataTypelIdentifier

```
enum class SensorDataTypelIdentifier [strong]
```

Enumerator

<code>LIGHT_LEVEL</code>	
<code>TEMPERATURE</code>	
<code>PRESSURE</code>	
<code>HUMIDITY</code>	
<code>MAG_FIELD_X</code>	
<code>MAG_FIELD_Y</code>	
<code>MAG_FIELD_Z</code>	
<code>GYRO_X</code>	
<code>GYRO_Y</code>	
<code>GYRO_Z</code>	
<code>ACCEL_X</code>	
<code>ACCEL_Y</code>	
<code>ACCEL_Z</code>	

Definition at line 18 of file `ISensor.h`.

8.94 ISensor.h

[Go to the documentation of this file.](#)

```

00001 // ISensor.h
00002 #ifndef ISENSOR_H
00003 #define ISENSOR_H
00004
00005 #include <cstdint>
00006 #include "hardware/i2c.h"
00007 #include <map>
00008 #include <string>
00009 #include <vector>
00010
00011 enum class SensorType {
00012     LIGHT,           // BH1750
00013     ENVIRONMENT,    // BME280
00014     MAGNETOMETER,   // HMC5883L
00015     IMU,             // MPU6050
00016 };
00017
00018 enum class SensorDataTypeIdentifier {
00019     LIGHT_LEVEL,
00020     TEMPERATURE,
00021     PRESSURE,
00022     HUMIDITY,
00023     MAG_FIELD_X,
00024     MAG_FIELD_Y,
00025     MAG_FIELD_Z,
00026     GYRO_X,
00027     GYRO_Y,
00028     GYRO_Z,
00029     ACCEL_X,
00030     ACCEL_Y,
00031     ACCEL_Z
00032 };
00033
00034 class ISensor {
00035 public:
00036     virtual ~ISensor() = default;
00037     virtual bool init() = 0;
00038     virtual float read_data(SensorDataTypeIdentifier type) = 0;
00039     virtual bool is_initialized() const = 0;
00040     virtual SensorType get_type() const = 0;
00041     virtual bool configure(const std::map<std::string, std::string>& config) = 0;
00042     virtual uint8_t get_address() const = 0;
00043 };
00044
00045 class SensorWrapper {
00046 public:
00047     static SensorWrapper& get_instance();
00048     bool sensor_init(SensorType type, i2c_inst_t* i2c = nullptr);
00049     bool sensor_configure(SensorType type, const std::map<std::string, std::string>& config);
00050     float sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType);
00051     ISensor* get_sensor(SensorType type);
00052
00053     std::vector<std::pair<SensorType, uint8_t>> scan_connected_sensors(i2c_inst_t* i2c);
00054     std::vector<std::pair<SensorType, uint8_t>> get_available_sensors();
00055
00056 private:
00057     std::map<SensorType, ISensor*> sensors;
00058     SensorWrapper();
00059 };
00060
00061 #endif // ISENSOR_H

```

8.95 lib/sensors/MPU6050/MPU6050.cpp File Reference

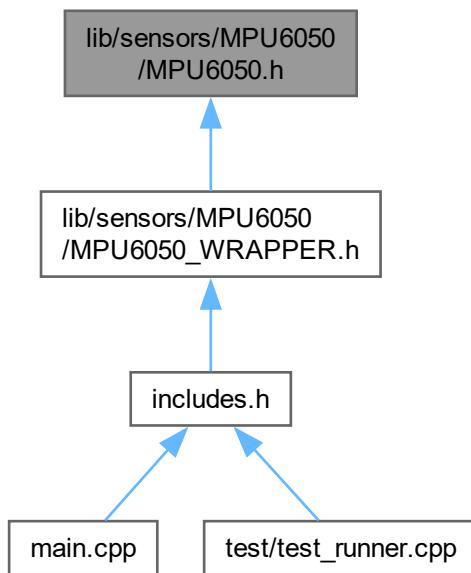
8.96 MPU6050.cpp

[Go to the documentation of this file.](#)

```
00001
```

8.97 lib/sensors/MPU6050/MPU6050.h File Reference

This graph shows which files directly or indirectly include this file:



8.98 MPU6050.h

[Go to the documentation of this file.](#)

00001

8.99 lib/sensors/MPU6050/MPU6050_WRAPPER.cpp File Reference

8.100 MPU6050_WRAPPER.cpp

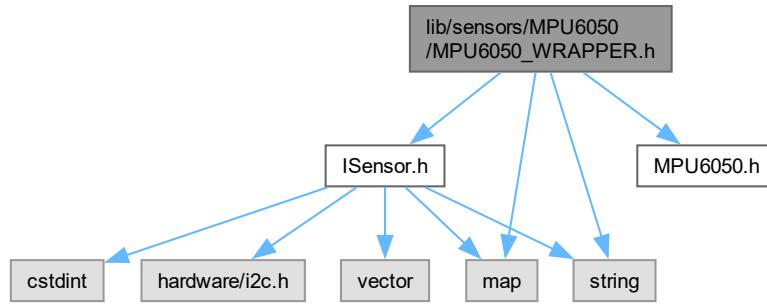
[Go to the documentation of this file.](#)

00001

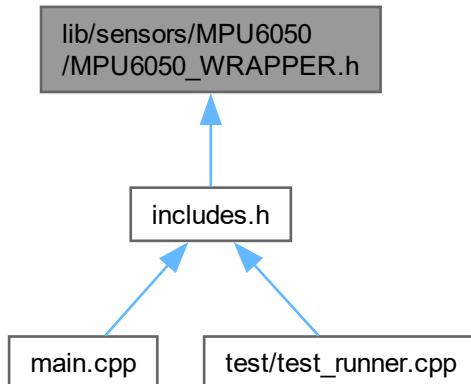
8.101 lib/sensors/MPU6050/MPU6050_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "MPU6050.h"
#include <map>
```

```
#include <string>
Include dependency graph for MPU6050_WRAPPER.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- class [MPU6050Wrapper](#)

8.102 MPU6050_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 "MPU6050.h"
00006 #include <map>
```

```

00007 #include <string>
00008
00009 class MPU6050Wrapper : public ISensor {
00010 private:
00011     MPU6050 sensor_;
00012     bool initialized_ = false;
00013
00014 public:
00015     MPU6050Wrapper();
00016
00017     bool init() override;
00018     float read_data(SensorDataTypeIdentifier type) override;
00019     bool is_initialized() const override;
00020     SensorType get_type() const override;
00021
00022     bool configure(const std::map<std::string, std::string>& config);
00023 };
00024
00025 #endif

```

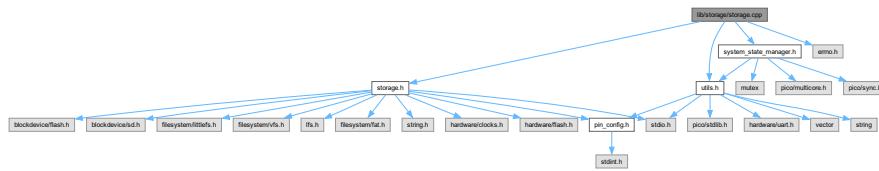
8.103 lib/storage/storage.cpp File Reference

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)**

8.103.1 Detailed Description

Implements file system operations for the Kubisat firmware.

This file contains functions for initializing the file system, opening, writing, reading, and closing files.

Definition in file [storage.cpp](#).

8.103.2 Function Documentation

8.103.2.1 `fs_init()`

```
bool fs_init (
    void )
```

Initializes the file system on the SD card.

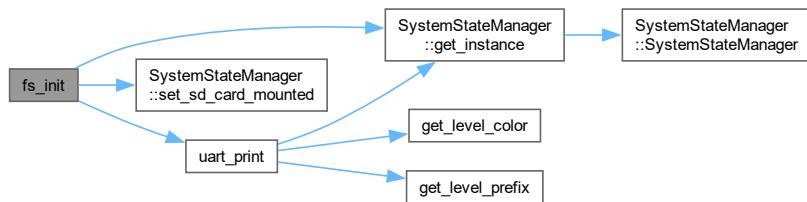
Returns

True if initialization was successful, false otherwise.

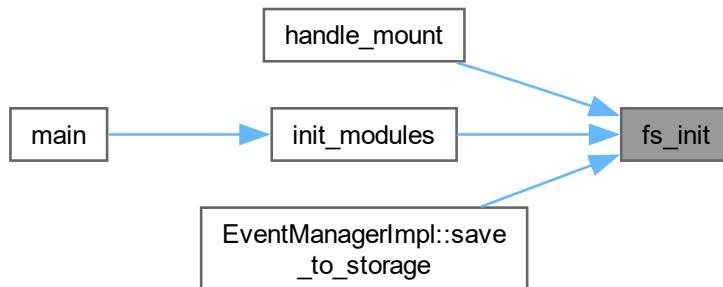
Mounts the littlefs file system on the SD card. If mounting fails, it formats the SD card with littlefs and then attempts to mount again.

Definition at line 25 of file [storage.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

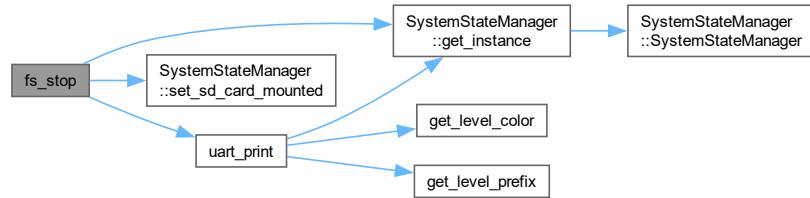


8.103.2.2 fs_stop()

```
bool fs_stop (
    void )
```

Definition at line 60 of file [storage.cpp](#).

Here is the call graph for this function:



8.104 storage.cpp

[Go to the documentation of this file.](#)

```
00001 /*
00002  * Copyright 2024, Hiroyuki OYAMA. All rights reserved.
00003  *
00004  * SPDX-License-Identifier: BSD-3-Clause
00005 */
00006 #include "storage.h"
00007 #include "errno.h"
00008 #include "utils.h"
00009 #include "system_state_manager.h"
00010
00017
00018
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
00068         return true;
00069     }

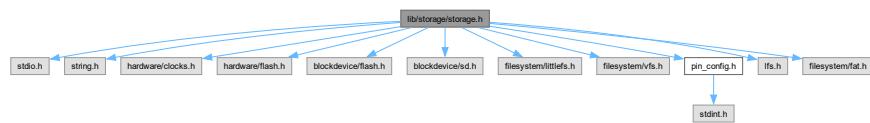
```

8.105 lib/storage/storage.h File Reference

```

#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:

```



Classes

- struct [FileHandle](#)

Functions

- [bool fs_init \(void\)](#)
Initializes the file system on the SD card.
- [FileHandle fs_open_file \(const char *filename, const char *mode\)](#)
- [ssize_t fs_write_file \(FileHandle &handle, const void *buffer, size_t size\)](#)
- [ssize_t fs_read_file \(FileHandle &handle, void *buffer, size_t size\)](#)
- [bool fs_close_file \(FileHandle &handle\)](#)
- [bool fs_file_exists \(const char *filename\)](#)

Variables

- bool `sd_card_mounted`

8.105.1 Function Documentation

8.105.1.1 `fs_init()`

```
bool fs_init (
    void )
```

Initializes the file system on the SD card.

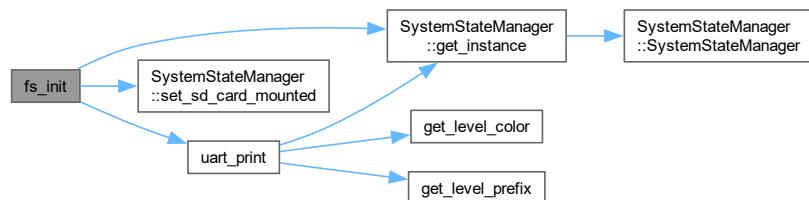
Returns

True if initialization was successful, false otherwise.

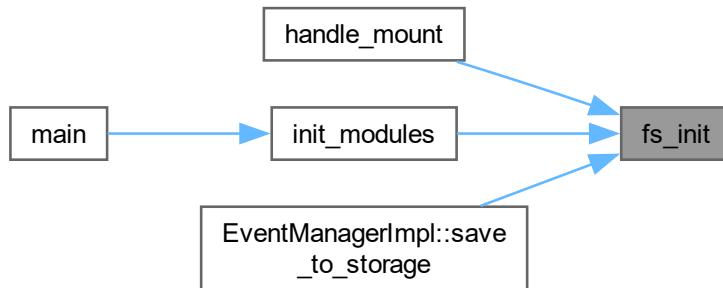
Mounts the littlefs file system on the SD card. If mounting fails, it formats the SD card with littlefs and then attempts to mount again.

Definition at line 25 of file [storage.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.105.1.2 `fs_open_file()`

```
FileHandle fs_open_file (
    const char * filename,
    const char * mode)
```

8.105.1.3 `fs_write_file()`

```
ssize_t fs_write_file (
    FileHandle & handle,
    const void * buffer,
    size_t size)
```

8.105.1.4 `fs_read_file()`

```
ssize_t fs_read_file (
    FileHandle & handle,
    void * buffer,
    size_t size)
```

8.105.1.5 `fs_close_file()`

```
bool fs_close_file (
    FileHandle & handle)
```

8.105.1.6 `fs_file_exists()`

```
bool fs_file_exists (
    const char * filename)
```

8.105.2 Variable Documentation

8.105.2.1 `sd_card_mounted`

```
bool sd_card_mounted [extern]
```

8.106 storage.h

[Go to the documentation of this file.](#)

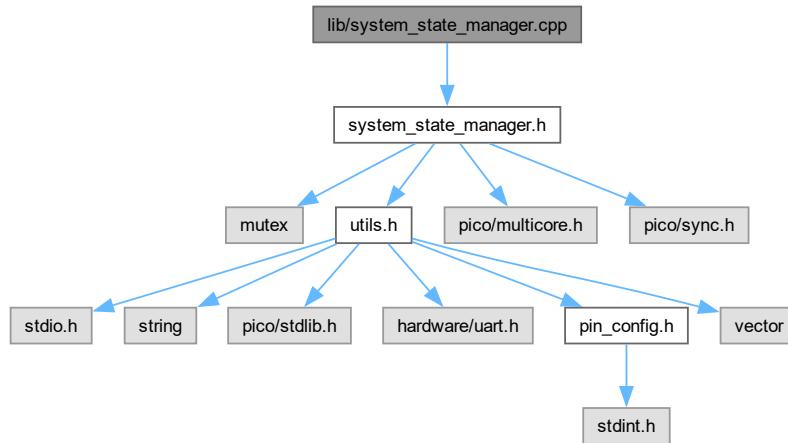
```

00001 #ifndef STORAGE_H
00002 #define STORAGE_H
00003
00004 #include <stdio.h>
00005 #include <string.h>
00006 #include <hardware/clocks.h>
00007 #include <hardware/flash.h>
00008 #include "blockdevice/flash.h"
00009 #include "blockdevice/sd.h"
00010 #include "filesystem/littlefs.h"
00011 #include "filesystem/vfs.h"
00012 #include "pin_config.h"
00013 #include "lfs.h"
00014 #include "filesystem/fat.h"
00015
00016
00017 extern bool sd_card_mounted;
00018
00019 struct FileHandle {
00020     int fd;
00021     bool is_open;
00022 };
00023
00024 bool fs_init(void);
00025 FileHandle fs_open_file(const char* filename, const char* mode);
00026 ssize_t fs_write_file(FileHandle& handle, const void* buffer, size_t size);
00027 ssize_t fs_read_file(FileHandle& handle, void* buffer, size_t size);
00028 bool fs_close_file(FileHandle& handle);
00029 bool fs_file_exists(const char* filename);
00030
00031 #endif
00032
00033 // void example_file_operations() {
00034 //     // Open a file for writing
00035 //     FileHandle log_file = fs_open_file("/log.txt", "w");
00036 //     if (!log_file.is_open) {
00037 //         uartPrint("Failed to open log file");
00038 //         return;
00039 //     }
00040
00041 //     // Write some data
00042 //     const char* message = "Hello, World!\n";
00043 //     ssize_t written = fs_write_file(log_file, message, strlen(message));
00044 //     if (written < 0) {
00045 //         uartPrint("Failed to write to log file");
00046 //     }
00047
00048 //     // Close the file
00049 //     fs_close_file(log_file);
00050
00051 //     // Open file for reading
00052 //     log_file = fs_open_file("/log.txt", "r");
00053 //     if (!log_file.is_open) {
00054 //         uartPrint("Failed to open log file for reading");
00055 //         return;
00056 //     }
00057
00058 //     // Read the data
00059 //     char buffer[128];
00060 //     ssize_t bytes_read = fs_read_file(log_file, buffer, sizeof(buffer) - 1);
00061 //     if (bytes_read > 0) {
00062 //         buffer[bytes_read] = '\0'; // Null terminate the string
00063 //         uartPrint(buffer);
00064 //     }
00065
00066 //     // Close the file
00067 //     fs_close_file(log_file);
00068 // }
```

8.107 lib/system_state_manager.cpp File Reference

Implementation of the [SystemStateManager](#) singleton class.

```
#include "system_state_manager.h"
Include dependency graph for system_state_manager.cpp:
```



8.107.1 Detailed Description

Implementation of the [SystemStateManager](#) singleton class.

Provides thread-safe implementation of system state management functions

Definition in file [system_state_manager.cpp](#).

8.108 system_state_manager.cpp

[Go to the documentation of this file.](#)

```

00001
00006
00007 #include "system_state_manager.h"
00008
00009 // Initialize static members
00010 SystemStateManager* SystemStateManager::instance = nullptr;
00011 mutex_t SystemStateManager::instance_mutex;
00012
00013 SystemStateManager& SystemStateManager::get_instance() {
00014     // Initialize mutex once
00015     static bool mutex_initialized = false;
00016     if (!mutex_initialized) {
00017         mutex_init(&instance_mutex);
00018         mutex_initialized = true;
00019     }
00020
00021     // Thread-safe singleton access
00022     mutex_enter_blocking(&instance_mutex);
00023     if (instance == nullptr) {
00024         instance = new SystemStateManager();
00025     }
00026     mutex_exit(&instance_mutex);
00027     return *instance;
00028 }
00029
00030 bool SystemStateManager::is_bootloader_reset_pending() const {
00031     return pending_bootloader_reset;
00032 }
00033
  
```

```

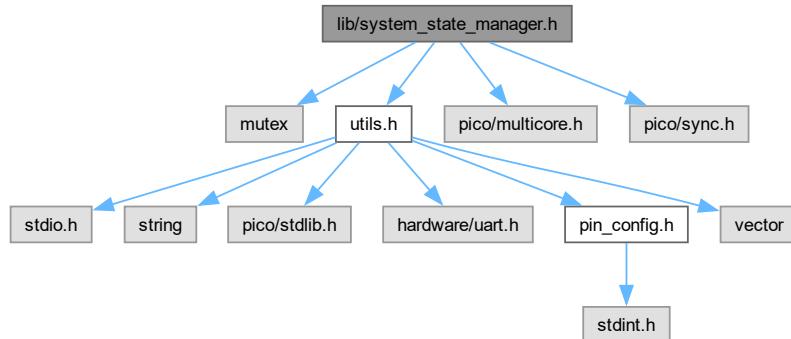
00034 void SystemStateManager::set_bootloader_reset_pending(bool pending) {
00035     pending_bootloader_reset = pending;
00036 }
00037
00038 bool SystemStateManager::is_gps_collection_paused() const {
00039     return gps_collection_paused;
00040 }
00041
00042 void SystemStateManager::set_gps_collection_paused(bool paused) {
00043     gps_collection_paused = paused;
00044 }
00045
00046 bool SystemStateManager::is_sd_card_mounted() const {
00047     return sd_card_mounted;
00048 }
00049
00050 void SystemStateManager::set_sd_card_mounted(bool mounted) {
00051     sd_card_mounted = mounted;
00052 }
00053
00054 VerbosityLevel SystemStateManager::get_uart_verbosity() const {
00055     return uart_verbosity;
00056 }
00057
00058 void SystemStateManager::set_uart_verbosity(VerbosityLevel level) {
00059     uart_verbosity = level;
00060 }

```

8.109 lib/system_state_manager.h File Reference

Declaration of the [SystemStateManager](#) singleton class for managing global system states.

```
#include <mutex>
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
Include dependency graph for system_state_manager.h:
```



This graph shows which files directly or indirectly include this file:



Classes

- class [SystemStateManager](#)

Singleton class for managing global system states.

8.109.1 Detailed Description

Declaration of the [SystemStateManager](#) singleton class for managing global system states.

The [SystemStateManager](#) provides a thread-safe way to manage system-wide states such as bootloader reset flags, GPS collection status, SD card mount status, and UART verbosity levels. It uses a singleton pattern to ensure a single instance across the system with controlled access through mutex protection.

Definition in file [system_state_manager.h](#).

8.110 system_state_manager.h

[Go to the documentation of this file.](#)

```

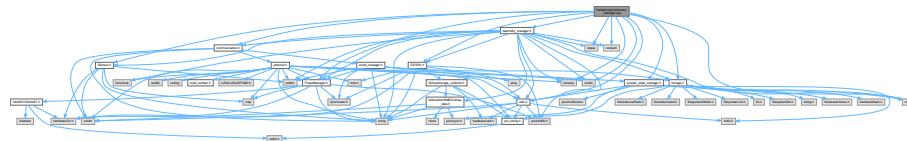
00001
00009
00010 #ifndef SYSTEM_STATE_MANAGER_H
00011 #define SYSTEM_STATE_MANAGER_H
00012
00013 #include <mutex>
00014 #include "utils.h"
00015 #include "pico/multicore.h"
00016 #include "pico/sync.h"
00017
00018 class SystemStateManager {
00019 private:
00020     static SystemStateManager* instance;
00021
00022     static mutex_t instance_mutex;
00023
00024     // Private states
00025     bool pending_bootloader_reset;
00026     bool gps_collection_paused;
00027     bool sd_card_mounted;
00028     VerbosityLevel uart_verbosity;
00029
00030     SystemStateManager() :
00031         pending_bootloader_reset(false),
00032         gps_collection_paused(false),
00033         sd_card_mounted(false),
00034         uart_verbosity(VerbosityLevel::DEBUG) {}
00035
00036 public:
00037     static SystemStateManager& get_instance();
00038
00039     bool is_bootloader_reset_pending() const;
00040
00041     void set_bootloader_reset_pending(bool pending);
00042
00043     bool is_gps_collection_paused() const;
00044
00045     void set_gps_collection_paused(bool paused);
00046
00047     bool is_sd_card_mounted() const;
00048
00049     void set_sd_card_mounted(bool mounted);
00050
00051     VerbosityLevel get_uart_verbosity() const;
00052
00053     void set_uart_verbosity(VerbosityLevel level);
00054
00055     // Delete copy constructor and assignment operator to enforce singleton pattern
00056     SystemStateManager(const SystemStateManager&) = delete;
00057     SystemStateManager& operator=(const SystemStateManager&) = delete;
00058
00059 };
00060
00061 #endif // SYSTEM_STATE_MANAGER_H

```

8.111 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 TELEMTRY_CSV_PATH "/telemetry.csv"`
Path to the telemetry CSV file on storage media.
- `#define SENSOR_DATA_CSV_PATH "/sensors.csv"`
Path to the sensor data CSV file on storage media.
- `#define DEFAULT_SAMPLE_INTERVAL_MS 1000`
Default interval between telemetry samples in milliseconds (2 seconds)
- `#define DEFAULT_FLUSH_THRESHOLD 10`
Default number of records to collect before flushing to storage.

Functions

- `bool telemetry_init ()`
Initialize the telemetry system.
- `bool collect_telemetry ()`
Collect telemetry data from sensors and power subsystems.
- `bool flush_telemetry ()`
Save buffered telemetry data to storage.
- `bool flush_sensor_data ()`
Save buffered sensor data to storage.
- `bool is_telemetry_collection_time (uint32_t current_time, uint32_t &last_collection_time)`
Check if it's time to collect telemetry based on interval.
- `bool is_telemetry_flush_time (uint32_t &collection_counter)`
Check if it's time to flush telemetry buffer based on count.
- `uint32_t get_telemetry_sample_interval ()`

- Get the current sample interval in milliseconds.
- void [set_telemetry_sample_interval](#) (uint32_t interval_ms)
Set the telemetry sample interval.
- uint32_t [get_telemetry_flush_threshold](#) ()
Get the number of records before flushing to storage.
- void [set_telemetry_flush_threshold](#) (uint32_t records)
Set the number of records before flushing to storage.
- std::string [get_last_telemetry_record_csv](#) ()
Gets the last telemetry record as a CSV string.
- std::string [get_last_sensor_record_csv](#) ()
Gets the last sensor data record as a CSV string.

Variables

- PowerManager powerManager
- DS3231 systemClock
- NMEAData nmea_data
- static uint32_t [sample_interval_ms](#) = [DEFAULT_SAMPLE_INTERVAL_MS](#)
Current sampling interval in milliseconds.
- static uint32_t [flush_threshold](#) = [DEFAULT_FLUSH_THRESHOLD](#)
Current flush threshold (number of records that triggers a flush)
- TelemetryRecord [telemetry_buffer](#) [[TELEMETRY_BUFFER_SIZE](#)]
Circular buffer for telemetry records.
- size_t [telemetry_buffer_count](#) = 0
- size_t [telemetry_buffer_write_index](#) = 0
- SensorDataRecord [sensor_data_buffer](#) [[TELEMETRY_BUFFER_SIZE](#)]
Circular buffer for sensor data records.
- size_t [sensor_data_buffer_count](#) = 0
- size_t [sensor_data_buffer_write_index](#) = 0
- mutex_t [telemetry_mutex](#)
Mutex for thread-safe access to the telemetry buffer.

8.111.1 Detailed Description

Implementation of telemetry collection and storage functionality.

Handles collecting, buffering, and persisting telemetry data from various satellite subsystems including power, sensors, and GPS

Definition in file [telemetry_manager.cpp](#).

8.111.2 Macro Definition Documentation

8.111.2.1 [TELEMETRY_CSV_PATH](#)

```
#define TELEMETRY_CSV_PATH "/telemetry.csv"
```

Path to the telemetry CSV file on storage media.

Definition at line 31 of file [telemetry_manager.cpp](#).

8.111.2.2 SENSOR_DATA_CSV_PATH

```
#define SENSOR_DATA_CSV_PATH "/sensors.csv"
```

Path to the sensor data CSV file on storage media.

Definition at line 36 of file [telemetry_manager.cpp](#).

8.111.2.3 DEFAULT_SAMPLE_INTERVAL_MS

```
#define DEFAULT_SAMPLE_INTERVAL_MS 1000
```

Default interval between telemetry samples in milliseconds (2 seconds)

Definition at line 41 of file [telemetry_manager.cpp](#).

8.111.2.4 DEFAULT_FLUSH_THRESHOLD

```
#define DEFAULT_FLUSH_THRESHOLD 10
```

Default number of records to collect before flushing to storage.

Definition at line 46 of file [telemetry_manager.cpp](#).

8.111.3 Variable Documentation

8.111.3.1 powerManager

```
PowerManager powerManager [extern]
```

8.111.3.2 systemClock

```
DS3231 systemClock [extern]
```

8.111.3.3 nmea_data

```
NMEAData nmea_data [extern]
```

Definition at line 3 of file [NMEA_data.cpp](#).

8.111.3.4 sample_interval_ms

```
uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS [static]
```

Current sampling interval in milliseconds.

Definition at line 51 of file [telemetry_manager.cpp](#).

8.111.3.5 flush_threshold

```
uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD [static]
```

Current flush threshold (number of records that triggers a flush)

Definition at line 56 of file [telemetry_manager.cpp](#).

8.112 telemetry_manager.cpp

[Go to the documentation of this file.](#)

```
00001 #include "telemetry_manager.h"
00002 #include "utils.h"
00003 #include "storage.h"
00004 #include "PowerManager.h"
00005 #include "ISensor.h"
00006 #include "DS3231.h"
00007 #include <deque>
00008 #include <mutex>
00009 #include <iomanip>
00010 #include <sstream>
00011 #include <cstdio>
00012 #include "communication.h"
00013 #include "system_state_manager.h"
00014
00021
00022
00023 extern PowerManager powerManager;
00024 extern DS3231 systemClock;
00025 extern NMEAData nmea_data;
00026
00027
00031 #define TELEMETRY_CSV_PATH "/telemetry.csv"
00032
00036 #define SENSOR_DATA_CSV_PATH "/sensors.csv"
00037
00041 #define DEFAULT_SAMPLE_INTERVAL_MS 1000
00042
00046 #define DEFAULT_FLUSH_THRESHOLD 10
00047
00051 static uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS;
00052
00056 static uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD;
00057
00061 TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE];
00062 size_t telemetry_buffer_count = 0;
00063 size_t telemetry_buffer_write_index = 0;
00064
00068 SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE];
00069 size_t sensor_data_buffer_count = 0;
00070 size_t sensor_data_buffer_write_index = 0;
00071
00075 mutex_t telemetry_mutex;
00076
00077 bool telemetry_init() {
00078     mutex_init(&telemetry_mutex);
00079
00080     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00081         bool success = true;
00082
00083         FILE* file = fopen(TELEMETRY_CSV_PATH, "w");
00084         if (!file) {
00085             file = fopen(TELEMETRY_CSV_PATH, "w");
00086             if (file) {
00087                 fprintf(file, "timestamp,build,battery_v,system_v,usb_ma,solar_ma,discharge_ma,"
00088                     "gps_time,latitude,lat_dir,longitude,lon_dir,speed_knots,course_deg,date,"
00089                     "fix_quality,satellites,altitude_m\n");
00090                 fclose(file);
00091                 uart_print("Created new telemetry log", VerbosityLevel::INFO);
00092             } else {
00093                 uart_print("Failed to create telemetry log", VerbosityLevel::ERROR);
00094                 success = false;
00095             }
00096         } else {
00097             fclose(file);
00098         }
00099     }
00100 }
```

```
00099     file = fopen(SENSOR_DATA_CSV_PATH, "w");
00100     if (!file) {
00101         file = fopen(SENSOR_DATA_CSV_PATH, "w");
00102         if (file) {
00103             fprintf(file, "timestamp,temperature,pressure,humidity,light\n");
00104             fclose(file);
00105             uart_print("Created new sensor data log", VerbosityLevel::INFO);
00106         } else {
00107             uart_print("Failed to create sensor data log", VerbosityLevel::ERROR);
00108             success = false;
00109         }
00110     }
00111     } else {
00112         fclose(file);
00113     }
00114
00115     return success;
00116 }
00117
00118 uart_print("Telemetry system initialized (storage not available)", VerbosityLevel::WARNING);
00119 return false;
00120 }
00121
00122 bool collect_telemetry() {
00123     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00124     uint32_t timestamp = systemClock.get_unix_time();
00125     TelemetryRecord record;
00126     record.timestamp = timestamp;
00127     record.build_version = std::to_string(BUILD_NUMBER);
00128
00129     // Power data
00130     record.battery_voltage = powerManager.get_voltage_battery();
00131     record.system_voltage = powerManager.get_voltage_5v();
00132     record.charge_current_usb = powerManager.get_current_charge_usb();
00133     record.charge_current_solar = powerManager.get_current_charge_solar();
00134     record.discharge_current = powerManager.get_current_draw();
00135
00136     // Get GPS RMC data
00137     std::vector<std::string> rmc_tokens = nmea_data.get_rmc_tokens();
00138     if (rmc_tokens.size() >= 12) { // RMC has at least 12 fields when complete
00139         record.time = rmc_tokens[1];
00140         record.latitude = rmc_tokens[3];
00141         record.lat_dir = rmc_tokens[4];
00142         record.longitude = rmc_tokens[5];
00143         record.lon_dir = rmc_tokens[6];
00144         record.speed = rmc_tokens[7];
00145         record.course = rmc_tokens[8];
00146         record.date = rmc_tokens[9];
00147     } else {
00148         // Fill with defaults if no GPS data
00149         record.time = "";
00150         record.latitude = "";
00151         record.lat_dir = "";
00152         record.longitude = "";
00153         record.lon_dir = "";
00154         record.speed = "";
00155         record.course = "";
00156         record.date = "";
00157     }
00158
00159     // Get GPS GGA data
00160     std::vector<std::string> gga_tokens = nmea_data.get_gga_tokens();
00161     if (gga_tokens.size() >= 15) { // GGA has 15 fields when complete
00162         record.fix_quality = gga_tokens[6];
00163         record.satellites = gga_tokens[7];
00164         record.altitude = gga_tokens[9];
00165     } else {
00166         // Fill with defaults if no GPS data
00167         record.fix_quality = "";
00168         record.satellites = "";
00169         record.altitude = "";
00170     }
00171
00172     SensorDataRecord sensor_record;
00173     sensor_record.timestamp = timestamp;
00174     sensor_record.temperature = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00175         SensorDataTypeIdentifier::TEMPERATURE);
00175     sensor_record.pressure = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00176         SensorDataTypeIdentifier::PRESSURE);
00176     sensor_record.humidity = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00177         SensorDataTypeIdentifier::HUMIDITY);
00177     sensor_record.light = sensor_wrapper.sensor_read_data(SensorType::LIGHT,
00178         SensorDataTypeIdentifier::LIGHT_LEVEL);
00178
00179     mutex_enter_blocking(&telemetry_mutex);
00180     telemetry_buffer[telemetry_buffer_write_index] = record;
```

```

00182     sensor_data_buffer[telemetry_buffer_write_index] = sensor_record;
00183     telemetry_buffer_write_index = (telemetry_buffer_write_index + 1) % TELEMETRY_BUFFER_SIZE;
00184     if (telemetry_buffer_count < TELEMETRY_BUFFER_SIZE) {
00185         telemetry_buffer_count++;
00186     }
00187
00188     mutex_exit(&telemetry_mutex);
00189
00190     uart_print("Telemetry collected", VerbosityLevel::DEBUG);
00191
00192     return true;
00193 }
00194
00195 bool flush_telemetry() {
00196     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00197         return false;
00198     }
00199
00200     mutex_enter_blocking(&telemetry_mutex);
00201
00203     if (telemetry_buffer_count == 0) {
00204         mutex_exit(&telemetry_mutex);
00205         return true; // Nothing to save
00206     }
00207
00208     FILE* file = fopen(TELEMETRY_CSV_PATH, "a");
00209     if (!file) {
00210         uart_print("Failed to open telemetry log for writing", VerbosityLevel::ERROR);
00211         mutex_exit(&telemetry_mutex);
00212         return false;
00213     }
00214
00215     // Calculate start index (for circular buffer)
00216     size_t read_index = 0;
00217     if (telemetry_buffer_count == TELEMETRY_BUFFER_SIZE) {
00218         // Buffer is full, start from oldest entry
00219         read_index = telemetry_buffer_write_index;
00220     }
00221
00222     // Write all records to CSV
00223     for (size_t i = 0; i < telemetry_buffer_count; i++) {
00224         fprintf(file, "%s\n", telemetry_buffer[read_index].to_csv().c_str());
00225         read_index = (read_index + 1) % TELEMETRY_BUFFER_SIZE;
00226     }
00227
00228     // Clear buffer after successful write
00229     telemetry_buffer_count = 0;
00230     telemetry_buffer_write_index = 0;
00231
00232     fclose(file);
00233
00234     mutex_exit(&telemetry_mutex);
00235     return true;
00236 }
00237
00238 bool flush_sensor_data() {
00239     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00240         return false;
00241     }
00242
00243     mutex_enter_blocking(&telemetry_mutex);
00244
00245     if (telemetry_buffer_count == 0) {
00246         mutex_exit(&telemetry_mutex);
00247         return true; // Nothing to save
00248     }
00249
00250     FILE* file = fopen(SENSOR_DATA_CSV_PATH, "a");
00251     if (!file) {
00252         uart_print("Failed to open sensor data log for writing", VerbosityLevel::ERROR);
00253         mutex_exit(&telemetry_mutex);
00254         return false;
00255     }
00256
00257     // Calculate start index (for circular buffer)
00258     size_t read_index = 0;
00259     if (telemetry_buffer_count == TELEMETRY_BUFFER_SIZE) {
00260         // Buffer is full, start from oldest entry
00261         read_index = telemetry_buffer_write_index;
00262     }
00263
00264     // Write all records to CSV
00265     for (size_t i = 0; i < telemetry_buffer_count; i++) {
00266         fprintf(file, "%s\n", sensor_data_buffer[read_index].to_csv().c_str());
00267         read_index = (read_index + 1) % TELEMETRY_BUFFER_SIZE;
00268     }

```

```
00269
00270 // We don't need to clear the buffer here since it's cleared in flush_telemetry()
00271 // and both buffers use the same indices
00272
00273 fclose(file);
00274
00275 mutex_exit(&telemetry_mutex);
00276 return true;
00277 }
00278
00279 bool is_telemetry_collection_time(uint32_t current_time, uint32_t& last_collection_time) {
00280     if (current_time - last_collection_time >= sample_interval_ms) {
00281         last_collection_time = current_time;
00282         return true;
00283     }
00284     return false;
00285 }
00286
00287 bool is_telemetry_flush_time(uint32_t& collection_counter) {
00288     if (collection_counter >= flush_threshold) {
00289         collection_counter = 0;
00290         return true;
00291     }
00292     return false;
00293 }
00294
00295 uint32_t get_telemetry_sample_interval() {
00296     return sample_interval_ms;
00297 }
00298
00299 void set_telemetry_sample_interval(uint32_t interval_ms) {
00300     if (interval_ms >= 100) { // Minimum 100ms
00301         sample_interval_ms = interval_ms;
00302     }
00303 }
00304
00305 uint32_t get_telemetry_flush_threshold() {
00306     return flush_threshold;
00307 }
00308
00309 void set_telemetry_flush_threshold(uint32_t records) {
00310     if (records >= 1 && records <= 100) {
00311         flush_threshold = records;
00312     }
00313 }
00314
00315 std::string get_last_telemetry_record_csv() {
00316     mutex_enter_blocking(&telemetry_mutex);
00317
00318     if (telemetry_buffer_count == 0) {
00319         mutex_exit(&telemetry_mutex);
00320         return "";
00321     }
00322
00323     size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) %
00324         TELEMETRY_BUFFER_SIZE;
00325
00326     TelemetryRecord last_record = telemetry_buffer[last_record_index];
00327
00328     mutex_exit(&telemetry_mutex);
00329
00330     return last_record.to_csv();
00331 }
00332
00333 std::string get_last_sensor_record_csv() {
00334     mutex_enter_blocking(&telemetry_mutex);
00335
00336     if (telemetry_buffer_count == 0) {
00337         mutex_exit(&telemetry_mutex);
00338         return "";
00339     }
00340
00341     size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) %
00342         TELEMETRY_BUFFER_SIZE;
00343
00344     SensorDataRecord last_record = sensor_data_buffer[last_record_index];
00345
00346     mutex_exit(&telemetry_mutex);
00347
00348     return last_record.to_csv();
00349 }
```

8.113 lib/telemetry/telemetry_manager.h File Reference

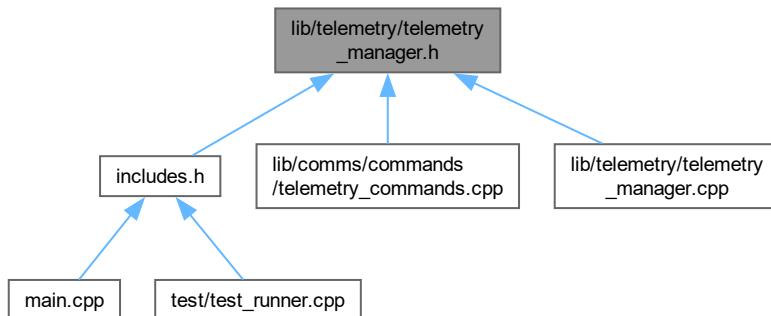
System telemetry collection and logging.

```
#include <cstdint>
#include <string>
#include "pico/stdlib.h"
#include "lib/location/NMEA/nmea_data.h"
#include "utils.h"
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include <functional>
```

Include dependency graph for telemetry_manager.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [TelemetryRecord](#)
Structure representing a single telemetry data point.
- struct [SensorDataRecord](#)

Functions

- `bool telemetry_init ()`
Initialize the telemetry system.
- `bool collect_telemetry ()`
Collect telemetry data from sensors and power subsystems.
- `bool flush_telemetry ()`
Save buffered telemetry data to storage.
- `bool flush_sensor_data ()`
Save buffered sensor data to storage.
- `bool is_telemetry_collection_time (uint32_t current_time, uint32_t &last_collection_time)`
Check if it's time to collect telemetry based on interval.
- `bool is_telemetry_flush_time (uint32_t &collection_counter)`
Check if it's time to flush telemetry buffer based on count.
- `uint32_t get_telemetry_sample_interval ()`
Get the current sample interval in milliseconds.
- `void set_telemetry_sample_interval (uint32_t interval_ms)`
Set the telemetry sample interval.
- `uint32_t get_telemetry_flush_threshold ()`
Get the number of records before flushing to storage.
- `void set_telemetry_flush_threshold (uint32_t records)`
Set the number of records before flushing to storage.
- `std::string get_last_telemetry_record_csv ()`
Gets the last telemetry record as a CSV string.
- `std::string get_last_sensor_record_csv ()`
Gets the last sensor data record as a CSV string.

Variables

- `constexpr int TELEMETRY_BUFFER_SIZE = 20`
Circular buffer for telemetry records.
- `TelemetryRecord telemetry_buffer [TELEMETRY_BUFFER_SIZE]`
Circular buffer for telemetry records.
- `size_t telemetry_buffer_count`
- `size_t telemetry_buffer_write_index`
- `SensorDataRecord sensor_data_buffer [TELEMETRY_BUFFER_SIZE]`
Circular buffer for sensor data records.
- `size_t sensor_data_buffer_count`
- `size_t sensor_data_buffer_write_index`
- `mutex_t telemetry_mutex`
Mutex for thread-safe access to the telemetry buffer.

8.113.1 Detailed Description

System telemetry collection and logging.

This module handles periodic collection and storage of telemetry data from various satellite subsystems including power management, sensors (temperature, pressure, humidity, light), and GPS data.

Telemetry is collected at configurable intervals and stored in a circular buffer before being flushed to persistent storage after a configurable number of records are collected.

Definition in file [telemetry_manager.h](#).

8.114 telemetry_manager.h

[Go to the documentation of this file.](#)

```

00001
00015
00016 #ifndef TELEMETRY_MANAGER_H
00017 #define TELEMETRY_MANAGER_H
00018
00019 #include <cstdint>
00020 #include <string>
00021 #include "pico/stdlib.h"
00022 #include "lib/location/NMEA/nmea_data.h"
00023 #include "utils.h"
00024 #include "storage.h"
00025 #include "PowerManager.h"
00026 #include "ISensor.h"
00027 #include "DS3231.h"
00028 #include <deque>
00029 #include <mutex>
00030 #include <iomanip>
00031 #include <sstream>
00032 #include <cstdio>
00033 #include "communication.h"
00034 #include <functional>
00035
00042 struct TelemetryRecord {
00043     uint32_t timestamp;
00044
00045     std::string build_version;
00046
00047     // Power data
00048     float battery_voltage;
00049     float system_voltage;
00050     float charge_current_usb;
00051     float charge_current_solar;
00052     float discharge_current;
00053
00054     // GPS data - key RMC fields
00055     std::string time;
00056     std::string latitude;
00057     std::string lat_dir;
00058     std::string longitude;
00059     std::string lon_dir;
00060     std::string speed;
00061     std::string course;
00062     std::string date;
00063
00064     // GPS data - key GGA fields
00065     std::string fix_quality;
00066     std::string satellites;
00067     std::string altitude;
00068
00069     // Convert record to CSV line
00070     std::string to_csv() const {
00071         std::stringstream ss;
00072         ss << timestamp << ","
00073             << build_version << ","
00074             << std::fixed << std::setprecision(3)
00075             << battery_voltage << ","
00076             << system_voltage << ","
00077             << charge_current_usb << ","
00078             << charge_current_solar << ","
00079             << discharge_current << ","
00080
00081             // GPS RMC data
00082             << time << ","
00083             << latitude << "," << lat_dir << ","
00084             << longitude << "," << lon_dir << ","
00085             << speed << ","
00086             << course << ","
00087             << date << ","
00088             // GPS GGA data
00089             << fix_quality << ","
00090             << satellites << ","
00091             << altitude;
00092
00093     return ss.str();
00094 }
00095 };
00096
00097
00098 struct SensorDataRecord {
00099     uint32_t timestamp;
00100     float temperature;
00101     float pressure;

```

```

00102     float humidity;
00103     float light;
00104
00105     // Convert record to CSV line
00106     std::string to_csv() const {
00107         std::stringstream ss;
00108         ss << timestamp << ","
00109         << std::fixed << std::setprecision(3)
00110         << temperature << ","
00111         << pressure << ","
00112         << humidity << ","
00113         << light;
00114     return ss.str();
00115 }
00116 };
00117
00118 constexpr int TELEMETRY_BUFFER_SIZE = 20;
00119 extern const int TELEMETRY_BUFFER_SIZE;
00120 extern TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE];
00121 extern size_t telemetry_buffer_count;
00122 extern size_t telemetry_buffer_write_index;
00123
00124 extern SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE];
00125 extern size_t sensor_data_buffer_count;
00126 extern size_t sensor_data_buffer_write_index;
00127
00128 extern mutex_t telemetry_mutex;
00129
00130
00131 bool telemetry_init();
00132
00133 bool collect_telemetry();
00134
00135 bool flush_telemetry();
00136
00137
00138 bool flush_sensor_data();
00139
00140 bool is_telemetry_collection_time(uint32_t current_time, uint32_t& last_collection_time);
00141
00142 bool is_telemetry_flush_time(uint32_t& collection_counter);
00143
00144 uint32_t get_telemetry_sample_interval();
00145
00146 void set_telemetry_sample_interval(uint32_t interval_ms);
00147
00148 uint32_t get_telemetry_flush_threshold();
00149
00150 void set_telemetry_flush_threshold(uint32_t records);
00151
00152
00153 std::string get_last_telemetry_record_csv();
00154
00155 std::string get_last_sensor_record_csv();
00156
00157 #endif // TELEMETRY_MANAGER_H
00158 // End of TelemetryManager group

```

8.115 lib/utils.cpp File Reference

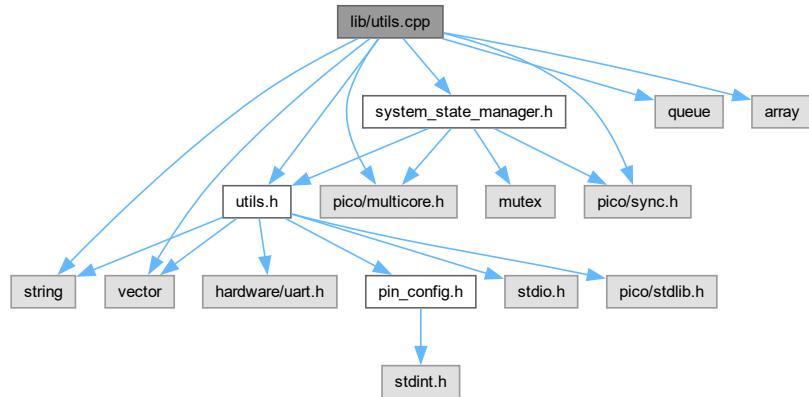
Implementation of utility functions for the Kubisat firmware.

```

#include "utils.h"
#include "pico/multicore.h"
#include "pico.sync.h"
#include <vector>
#include <queue>
#include <string>
#include <array>
#include "system_state_manager.h"

```

Include dependency graph for utils.cpp:



Functions

- `std::string get_level_color (VerbosityLevel level)`
Gets ANSI color code for verbosity level.
- `std::string get_level_prefix (VerbosityLevel level)`
Gets text prefix for verbosity level.
- `void uart_print (const std::string &msg, VerbosityLevel level, uart_inst_t *uart)`
Prints a message to the UART with a timestamp and core number.

Variables

- `static mutex_t uart_mutex`
Mutex for UART access protection.

8.115.1 Detailed Description

Implementation of utility functions for the Kubisat firmware.

Definition in file [utils.cpp](#).

8.115.2 Function Documentation

8.115.2.1 `get_level_color()`

```
std::string get_level_color (
    VerbosityLevel level)
```

Gets ANSI color code for verbosity level.

Parameters

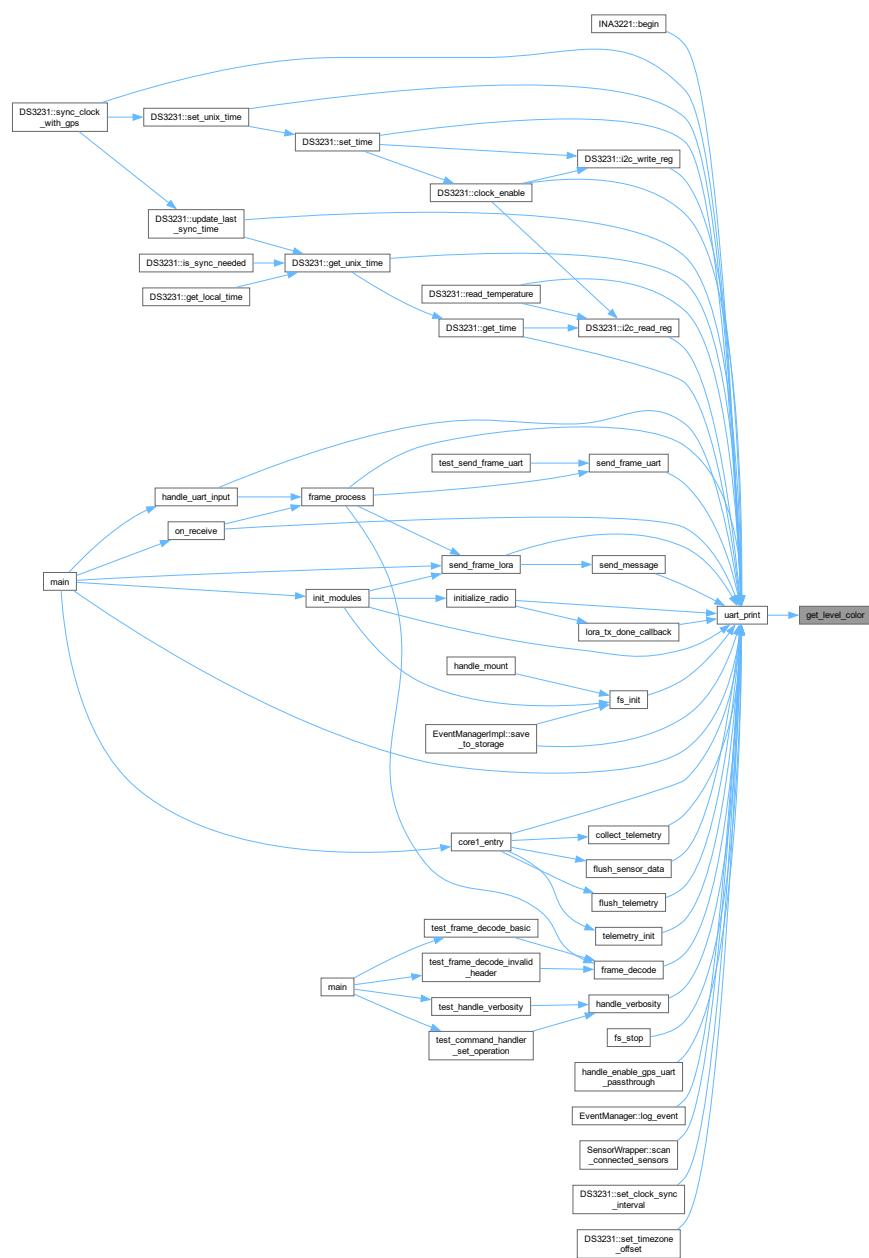
<i>level</i>	The verbosity level
--------------	---------------------

Returns

ANSI color escape sequence

Definition at line 25 of file [utils.cpp](#).

Here is the caller graph for this function:



8.115.2.2 `get_level_prefix()`

```
std::string get_level_prefix (
    VerbosityLevel level)
```

Gets text prefix for verbosity level.

Parameters

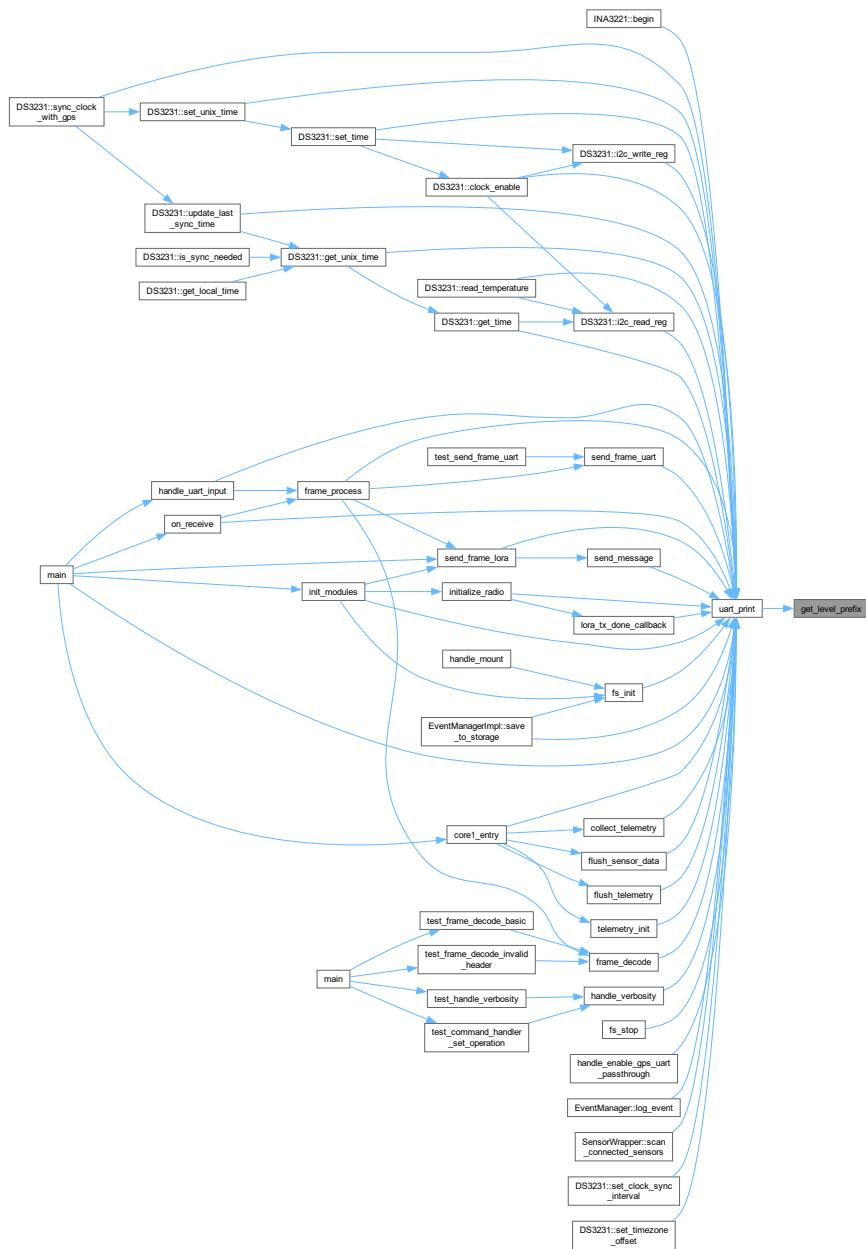
<i>level</i>	The verbosity level
--------------	---------------------

Returns

Text prefix for the level

Definition at line [41](#) of file [utils.cpp](#).

Here is the caller graph for this function:



8.115.2.3 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.

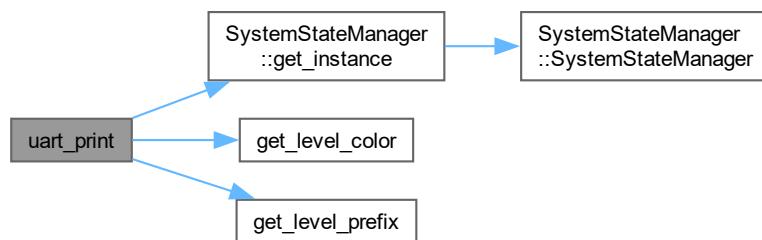
Parameters

<i>msg</i>	The message to print.
<i>logToFile</i>	A flag indicating whether to log the message to a file (currently not implemented).
<i>uart</i>	The UART instance to use for printing.

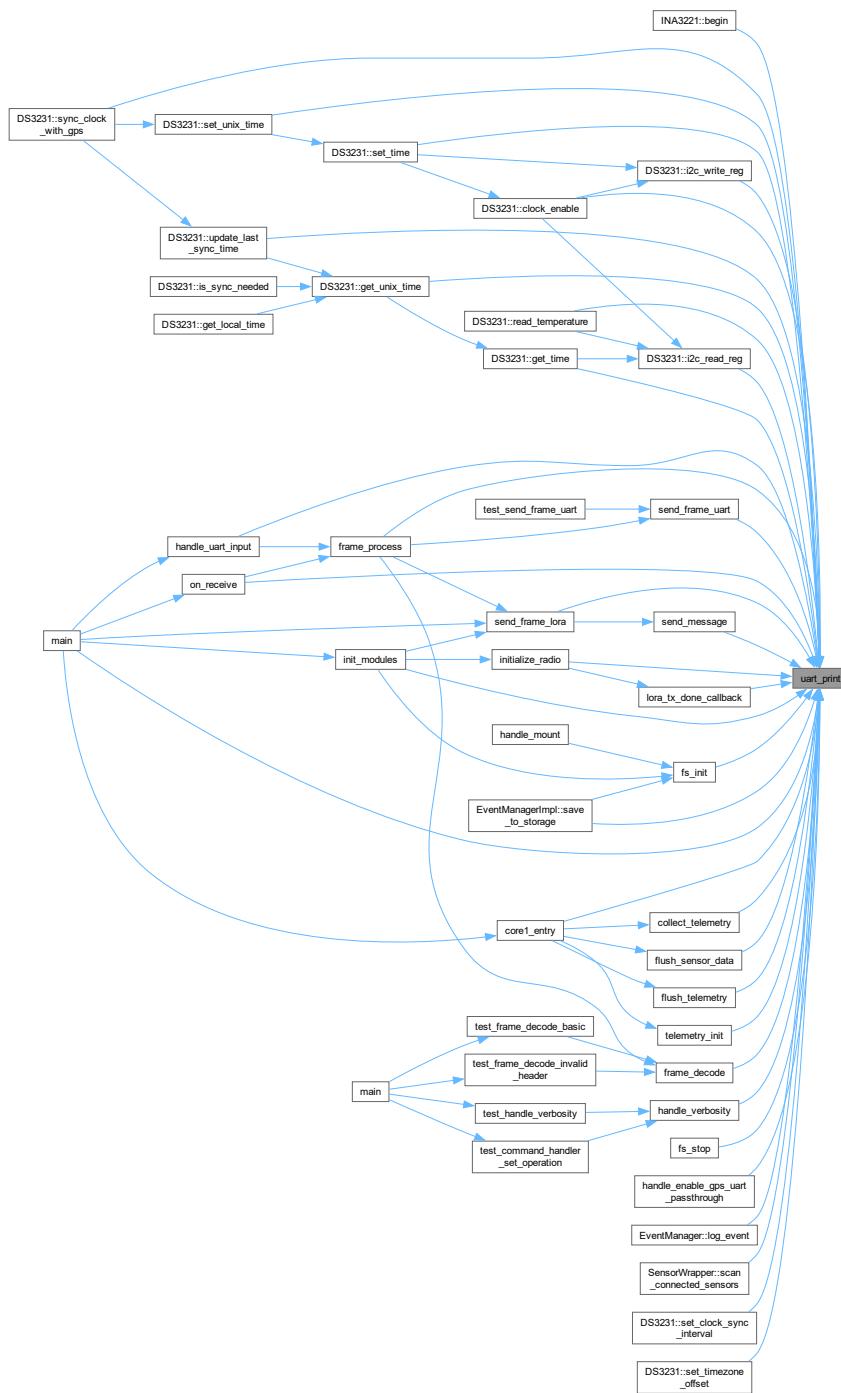
Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 59 of file [utils.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.115.3 Variable Documentation

8.115.3.1 uart_mutex

```
mutex_t uart_mutex [static]
```

Mutex for UART access protection.

Definition at line 17 of file [utils.cpp](#).

8.116 utils.cpp

[Go to the documentation of this file.](#)

```

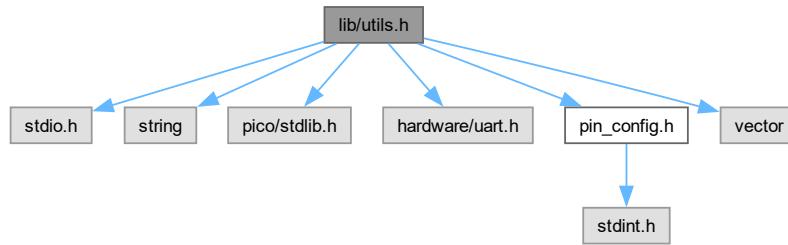
00001 #include "utils.h"
00002 #include "pico/multicore.h"
00003 #include "pico/sync.h"
00004 #include <vector>
00005 #include <queue>
00006 #include <string>
00007 #include <array>
00008 #include "system_state_manager.h"
00009
00014
00015
00017 static mutex_t uart_mutex;
00018
00019
00025 std::string get_level_color(VerbosityLevel level) {
00026     switch (level) {
00027         case VerbosityLevel::ERROR:    return ANSI_RED;
00028         case VerbosityLevel::WARNING: return ANSI_YELLOW;
00029         case VerbosityLevel::INFO:    return ANSI_GREEN;
00030         case VerbosityLevel::DEBUG:   return ANSI_BLUE;
00031         default:                     return "";
00032     }
00033 }
00034
00035
00041 std::string get_level_prefix(VerbosityLevel level) {
00042     switch (level) {
00043         case VerbosityLevel::ERROR:    return "ERROR: ";
00044         case VerbosityLevel::WARNING: return "WARNING: ";
00045         case VerbosityLevel::INFO:    return "INFO: ";
00046         case VerbosityLevel::DEBUG:   return "DEBUG: ";
00047         default:                     return "";
00048     }
00049 }
00050
00059 void uart_print(const std::string& msg, VerbosityLevel level, uart_inst_t* uart) {
00060     if (static_cast<int>(level) >
00061         static_cast<int>(SystemStateManager::get_instance().get_uart_verbosity())) {
00062         return;
00063     }
00064     static bool mutex_initiated = false;
00065     if (!mutex_initiated) {
00066         mutex_init(&uart_mutex);
00067         mutex_initiated = true;
00068     }
00069     uint32_t timestamp = to_ms_since_boot(get_absolute_time());
00070     uint core_num = get_core_num();
00071
00073     std::string color = get_level_color(level);
00074     std::string prefix = get_level_prefix(level);
00075     std::string msg_to_send = "[" + std::to_string(timestamp) + "ms] - Core " +
00076                             std::to_string(core_num) + ":" + color + prefix + ANSI_RESET + msg + "\r\n";
00077
00079     mutex_enter_blocking(&uart_mutex);
00080     uart_puts(uart, msg_to_send.c_str());
00081     mutex_exit(&uart_mutex);
00082 }
```

8.117 lib/utils.h File Reference

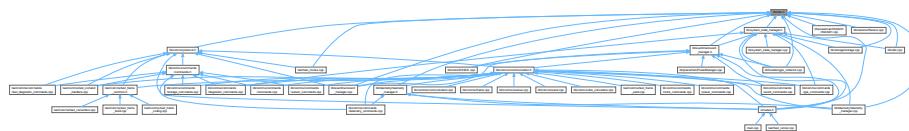
Utility functions and definitions for the Kubisat firmware.

```
#include <stdio.h>
#include <string>
#include "pico/stl.h"
#include "hardware/uart.h"
#include "pin_config.h"
```

```
#include <vector>
Include dependency graph for utils.h:
```



This graph shows which files directly or indirectly include this file:



Macros

- `#define ANSI_RED "\033[31m"`
ANSI escape codes for terminal color output.
- `#define ANSI_GREEN "\033[32m"`
- `#define ANSI_YELLOW "\033[33m"`
- `#define ANSI_BLUE "\033[34m"`
- `#define ANSI_RESET "\033[0m"`

Enumerations

- enum class `VerbosityLevel` {
 `SILENT = 0` , `ERROR = 1` , `WARNING = 2` , `INFO = 3` ,
 `DEBUG = 4` }

Verbosity levels for logging system.

Functions

- void `uart_print` (const std::string &msg, `VerbosityLevel` level=`VerbosityLevel::INFO`, `uart_inst_t *uart=DEBUG_UART_PORT`)
Prints a message to UART with timestamp and formatting.

8.117.1 Detailed Description

Utility functions and definitions for the Kabisat firmware.

Contains UART logging, color definitions, and CRC calculations

Definition in file [utils.h](#).

8.117.2 Macro Definition Documentation

8.117.2.1 ANSI_RED

```
#define ANSI_RED "\033[31m"
```

ANSI escape codes for terminal color output.

Definition at line [20](#) of file [utils.h](#).

8.117.2.2 ANSI_GREEN

```
#define ANSI_GREEN "\033[32m"
```

Definition at line [21](#) of file [utils.h](#).

8.117.2.3 ANSI_YELLOW

```
#define ANSI_YELLOW "\033[33m"
```

Definition at line [22](#) of file [utils.h](#).

8.117.2.4 ANSI_BLUE

```
#define ANSI_BLUE "\033[34m"
```

Definition at line [23](#) of file [utils.h](#).

8.117.2.5 ANSI_RESET

```
#define ANSI_RESET "\033[0m"
```

Definition at line [24](#) of file [utils.h](#).

8.117.3 Enumeration Type Documentation

8.117.3.1 VerbosityLevel

```
enum class VerbosityLevel [strong]
```

Verbosity levels for logging system.

Enumerator

SILENT	No output
ERROR	Only critical errors
WARNING	Warnings and errors
INFO	Normal operation information
DEBUG	Detailed debug information

Definition at line [30](#) of file [utils.h](#).

8.117.4 Function Documentation

8.117.4.1 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>logToFile</i>	Whether to store the message in log storage
<i>uart</i>	The UART port to use

Prints a message to UART with timestamp and formatting.

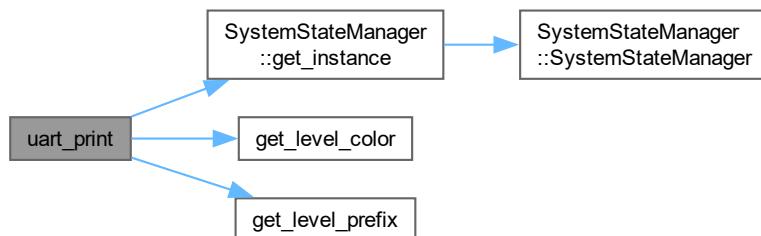
Parameters

<i>msg</i>	The message to print.
<i>logToFile</i>	A flag indicating whether to log the message to a file (currently not implemented).
<i>uart</i>	The UART instance to use for printing.

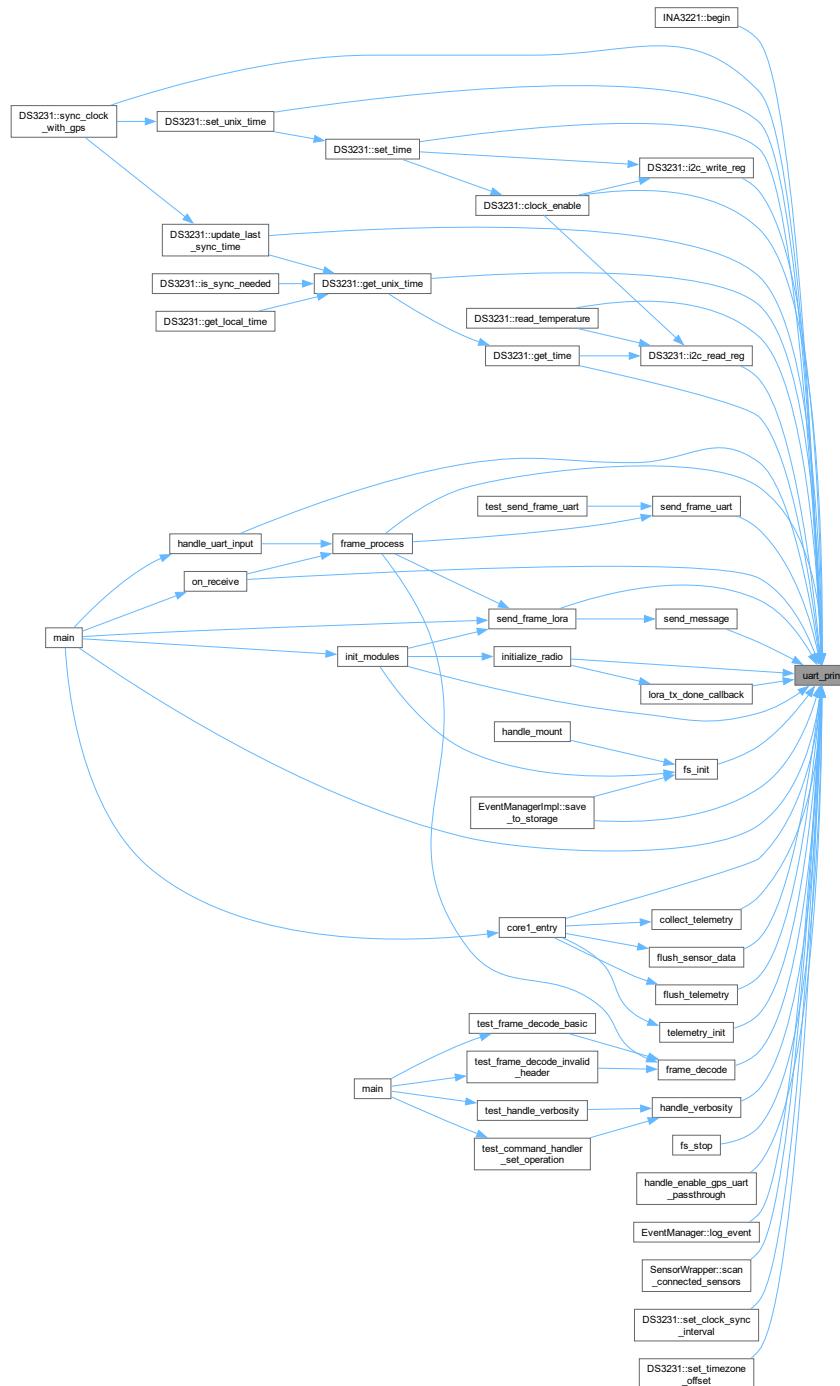
Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 59 of file [utils.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.118 utils.h

[Go to the documentation of this file.](#)

```
00001 #ifndef UTILS_H
00002 #define UTILS_H
00003
00004 #include <stdio.h>
```

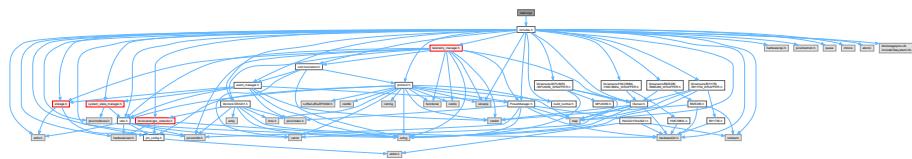
```

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
00019 #define ANSI_RED      "\033[31m"
00020 #define ANSI_GREEN    "\033[32m"
00021 #define ANSI_YELLOW   "\033[33m"
00022 #define ANSI_BLUE     "\033[34m"
00023 #define ANSI_RESET    "\033[0m"
00024
00025
00026
00027 enum class VerboseLevel {
00028     SILENT = 0,
00029     ERROR = 1,
00030     WARNING = 2,
00031     INFO = 3,
00032     DEBUG = 4
00033 };
00034
00035
00036
00037
00038
00039 void uart_print(const std::string& msg,
00040                   VerboseLevel level = VerboseLevel::INFO,
00041                   uart_inst_t* uart = DEBUG_UART_PORT);
00042
00043
00044
00045
00046
00047
00048
00049
00050
00051 #endif

```

8.119 main.cpp File Reference

#include "includes.h"
Include dependency graph for main.cpp:



Macros

- #define LOG_FILENAME "/log.txt"

Functions

- void core1_entry ()
- bool init_pico_hw ()
- bool init_modules ()
- int main ()

Variables

- PowerManager powerManager (MAIN_I2C_PORT)
- DS3231 systemClock (MAIN_I2C_PORT)
- char buffer [BUFFER_SIZE] = {0}
- int buffer_index = 0

8.119.1 Macro Definition Documentation

8.119.1.1 LOG_FILENAME

```
#define LOG_FILENAME "/log.txt"
```

Definition at line 3 of file [main.cpp](#).

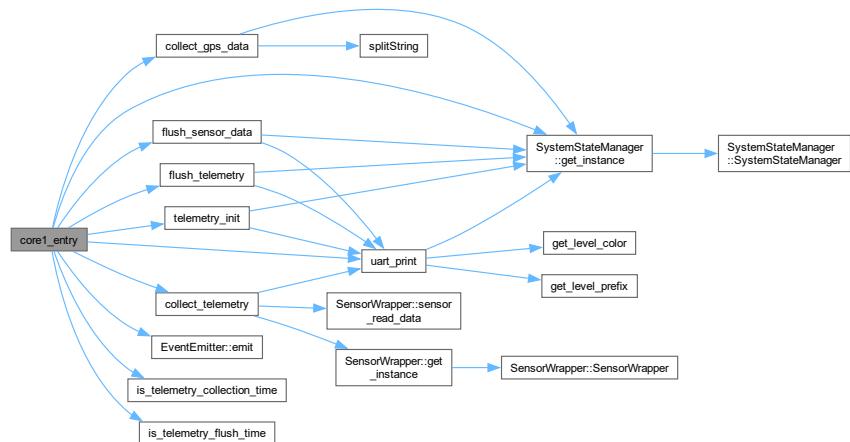
8.119.2 Function Documentation

8.119.2.1 core1_entry()

```
void core1_entry ()
```

Definition at line 11 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.119.2.2 init_pico_hw()

```
bool init_pico_hw ()
```

Definition at line 56 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

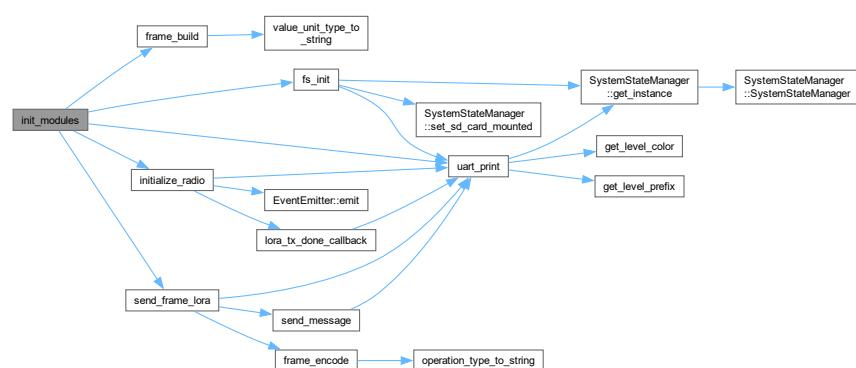


8.119.2.3 init_modules()

```
bool init_modules ()
```

Definition at line 93 of file [main.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

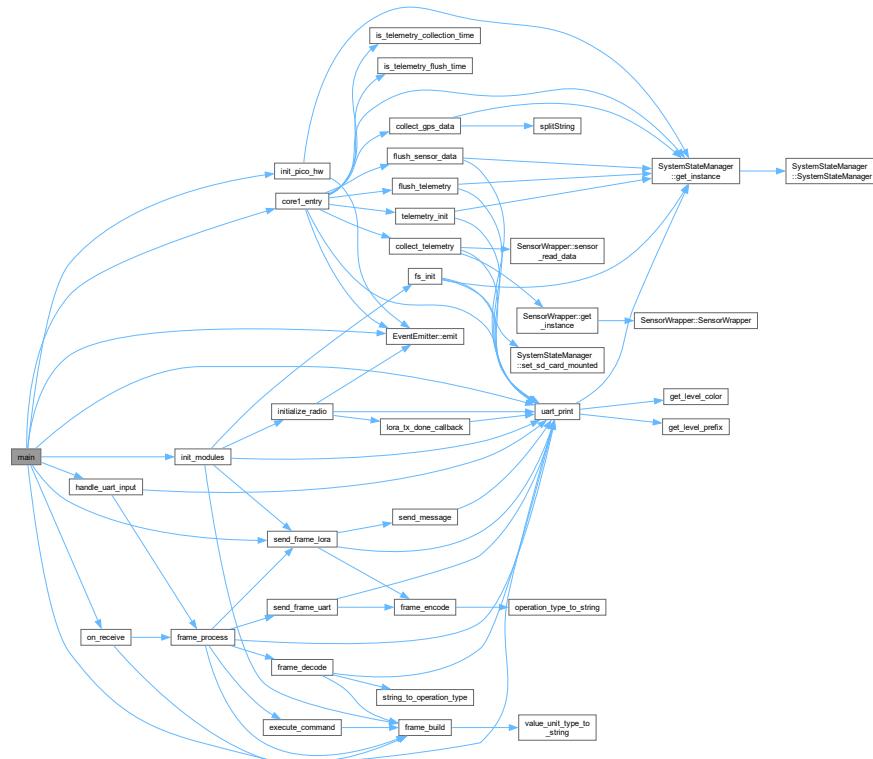


8.119.2.4 main()

```
int main (
    void )
```

Definition at line 149 of file [main.cpp](#).

Here is the call graph for this function:



8.119.3 Variable Documentation

8.119.3.1 powerManager

```
PowerManager powerManager(MAIN_I2C_PORT) (
    MAIN_I2C_PORT )
```

8.119.3.2 systemClock

```
DS3231 systemClock(MAIN_I2C_PORT) (
    MAIN_I2C_PORT )
```

8.119.3.3 buffer

```
char buffer[BUFFER_SIZE] = {0}
```

Definition at line 8 of file [main.cpp](#).

8.119.3.4 buffer_index

```
int buffer_index = 0
```

Definition at line 9 of file [main.cpp](#).

8.120 main.cpp

[Go to the documentation of this file.](#)

```
00001 #include "includes.h"
00002
00003 #define LOG_FILENAME "/log.txt"
00004
00005 PowerManager powerManager(MAIN_I2C_PORT);
00006 DS3231 systemClock(MAIN_I2C_PORT);
00007
00008 char buffer[BUFFER_SIZE] = {0};
00009 int buffer_index = 0;
00010
00011 void core1_entry() {
00012     uart_print("Starting core 1", VerbosityLevel::DEBUG);
00013     EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::CORE1_START);
00014
00015     uint32_t last_clock_check_time = 0;
00016     uint32_t last telemetry_time = 0;
00017     uint32_t telemetry_collection_counter = 0;
00018
00019     telemetry_init();
00020
00021     while (true) {
00022         collect_gps_data();
00023
00024         uint32_t currentTime = to_ms_since_boot(get_absolute_time());
00025
00026         uint32_t check_interval_ms = systemClock.get_clock_sync_interval() * 60000;
00027         if (currentTime - last_clock_check_time >= check_interval_ms) {
00028             last_clock_check_time = currentTime;
00029
00030             if (systemClock.is_sync_needed()) {
00031                 uart_print("Clock sync interval reached, attempting sync", VerbosityLevel::INFO);
00032                 systemClock.sync_clock_with_gps();
00033             }
00034         }
00035
00036         if (is_telemetry_collection_time(currentTime, last telemetry_time)) {
00037             collect_telemetry();
00038             telemetry_collection_counter++;
00039
00040             if (is_telemetry_flush_time(telemetry_collection_counter)) {
00041                 flush_telemetry();
00042                 flush_sensor_data();
00043             }
00044         }
00045
00046         if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
00047             sleep_ms(100);
00048         }
00049     }
00050 }
```

```

00048         uart_print("Entering BOOTSEL mode...", VerbosityLevel::WARNING);
00049         reset_usb_boot(0, 0);
00050     }
00051
00052     sleep_ms(10);
00053 }
00055
00056 bool init_pico_hw() {
00057     stdio_init_all();
00058
00059     uart_init(DEBUG_UART_PORT, DEBUG_UART_BAUD_RATE);
00060     gpio_set_function(DEBUG_UART_TX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_TX_PIN));
00061     gpio_set_function(DEBUG_UART_RX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_RX_PIN));
00062
00063     uart_init(GPS_UART_PORT, GPS_UART_BAUD_RATE);
00064     gpio_set_function(GPS_UART_TX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_TX_PIN));
00065     gpio_set_function(GPS_UART_RX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_RX_PIN));
00066
00067     gpio_init(PICO_DEFAULT_LED_PIN);
00068     gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
00069
00070     gpio_init(PICO_DEFAULT_LED_PIN);
00071     gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
00072     gpio_put(PICO_DEFAULT_LED_PIN, 1);
00073
00074     i2c_init(MAIN_I2C_PORT, 400 * 1000);
00075     gpio_set_function(MAIN_I2C_SCL_PIN, GPIO_FUNC_I2C);
00076     gpio_set_function(MAIN_I2C_SDA_PIN, GPIO_FUNC_I2C);
00077     gpio_pull_up(MAIN_I2C_SCL_PIN);
00078     gpio_pull_up(MAIN_I2C_SDA_PIN);
00079
00080     gpio_init(GPS_POWER_ENABLE_PIN);
00081     gpio_set_dir(GPS_POWER_ENABLE_PIN, GPIO_OUT);
00082     gpio_put(GPS_POWER_ENABLE_PIN, 1);
00083
00084     SystemStateManager::get_instance();
00085
00086     EventEmitter::emit(EventGroup::GPS, GPSEvent::POWER_ON);
00087
00088     system("color");
00089
00090     return true;
00091 }
00092
00093 bool init_modules(){
00094     bool radio_init_status = false;
00095     radio_init_status = initialize_radio();
00096
00097     bool sd_init_status = fs_init();
00098     if (sd_init_status) {
00099         FILE *fp = fopen(LOG_FILENAME, "w");
00100         if (fp) {
00101             uart_print("Log file opened.", VerbosityLevel::DEBUG);
00102             int bytes_written = fprintf(fp, "System init started.\n");
00103             uart_print("Written " + std::to_string(bytes_written) + " bytes.", VerbosityLevel::DEBUG);
00104             int close_status = fclose(fp);
00105             uart_print("Close file status: " + std::to_string(close_status), VerbosityLevel::DEBUG);
00106
00107             struct stat file_stat;
00108             if (stat(LOG_FILENAME, &file_stat) == 0) {
00109                 size_t file_size = file_stat.st_size;
00110                 uart_print("File size: " + std::to_string(file_size) + " bytes",
00111                         VerbosityLevel::DEBUG);
00112             } else {
00113                 uart_print("Failed to get file size", VerbosityLevel::ERROR);
00114             }
00115             uart_print("File path: /" + std::string(LOG_FILENAME), VerbosityLevel::DEBUG);
00116         } else {
00117             uart_print("Failed to open log file for writing.", VerbosityLevel::ERROR);
00118         }
00119     }
00120
00121     if (sd_init_status) {
00122         uart_print("SD card init: OK", VerbosityLevel::DEBUG);
00123     } else {
00124         uart_print("SD card init: FAILED", VerbosityLevel::ERROR);
00125     }
00126
00127     if (radio_init_status) {
00128         uart_print("Radio init: OK", VerbosityLevel::DEBUG);
00129     } else {
00130         uart_print("Radio init: FAILED", VerbosityLevel::ERROR);
00131     }
00132
00133     Frame boot = frame_build(OperationType::RES, 0, 0, "HELLO");

```

```

00134     send_frame_lora/boot);
00135
00136     // uart_print("Initializing sensors...", VerbosityLevel::DEBUG);
00137     // SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00138     // bool light_sensor_init = sensor_wrapper.sensor_init(SensorType::LIGHT, MAIN_I2C_PORT);
00139     // bool env_sensor_init = sensor_wrapper.sensor_init(SensorType::ENVIRONMENT, MAIN_I2C_PORT);
00140     // bool mag_sensor_init = sensor_wrapper.sensor_init(SensorType::MAGNETOMETER, MAIN_I2C_PORT);
00141
00142     // if (!light_sensor_init || !env_sensor_init || !mag_sensor_init) {
00143     //     uart_print("One or more sensors failed to initialize", VerbosityLevel::WARNING);
00144     // }
00145
00146     return sd_init_status && radio_init_status;
00147 }
00148
00149 int main()
00150 {
00151     init_pico_hw();
00152     sleep_ms(100);
00153     init_modules();
00154     EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::BOOT);
00155     sleep_ms(100);
00156     multicore_launch_core1(core1_entry);
00157
00158     gpio_put(PICO_DEFAULT_LED_PIN, 0);
00159
00160     bool power_manager_init_status = powerManager.initialize();
00161     if (power_manager_init_status)
00162     {
00163         std::map<std::string, std::string> power_config = {
00164             {"operating_mode", "continuous"},
00165             {"averaging_mode", "16"},
00166         };
00167         powerManager.configure(power_config);
00168     } else {
00169         uart_print("Power manager init error", VerbosityLevel::ERROR);
00170     }
00171
00172     Frame boot = frame_build(OperationType::RES, 0, 0, "START");
00173     send_frame_lora/boot);
00174
00175     std::string boot_string = "System init completed @ " +
00176     std::to_string(to_ms_since_boot(get_absolute_time())) + " ms";
00177     uart_print(boot_string, VerbosityLevel::WARNING);
00178
00179     gpio_put(PICO_DEFAULT_LED_PIN, 1);
00180
00181     while (true)
00182     {
00183         int packet_size = LoRa.parse_packet();
00184         if (packet_size)
00185         {
00186             on_receive(packet_size);
00187         }
00188         handle_uart_input();
00189     }
00190
00191     return 0;
00192 }
```

8.121 test/comms/commands/test_clock_commands.cpp File Reference

8.122 test_clock_commands.cpp

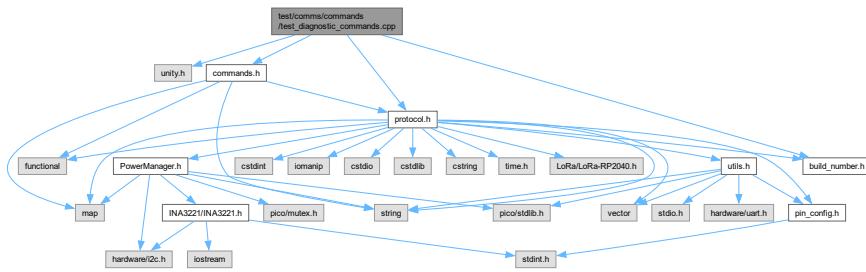
[Go to the documentation of this file.](#)

00001

8.123 test/comms/commands/test_diagnostic_commands.cpp File Reference

```
#include "unity.h"
#include "commands.h"
```

```
#include "protocol.h"
#include "build_number.h"
Include dependency graph for test_diagnostic_commands.cpp:
```



Functions

- void `test_handle_get_commands_list` (void)
- void `test_handle_get_build_version` (void)
- void `test_handle_verbosity` (void)
- void `test_handle_enter_bootloader_mode` (void)

8.123.1 Function Documentation

8.123.1.1 `test_handle_get_commands_list()`

```
void test_handle_get_commands_list (
    void )
```

Definition at line 6 of file `test_diagnostic_commands.cpp`.

Here is the call graph for this function:



Here is the caller graph for this function:

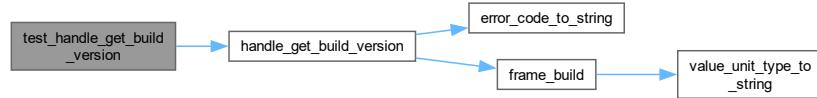


8.123.1.2 test_handle_get_build_version()

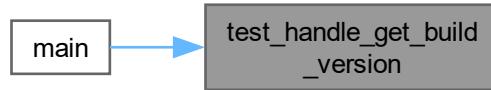
```
void test_handle_get_build_version (
    void )
```

Definition at line 15 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

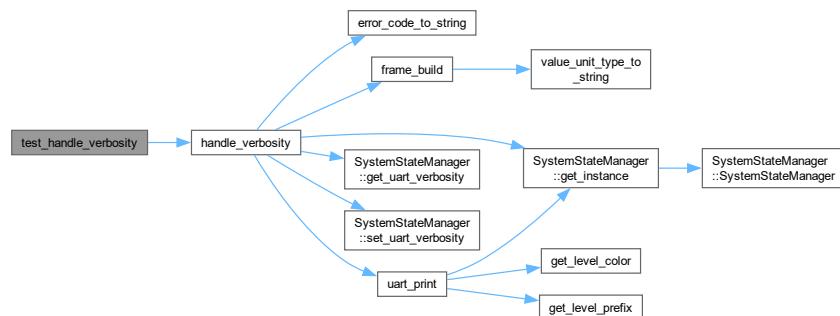


8.123.1.3 test_handle_verbosity()

```
void test_handle_verbosity (
    void )
```

Definition at line 25 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

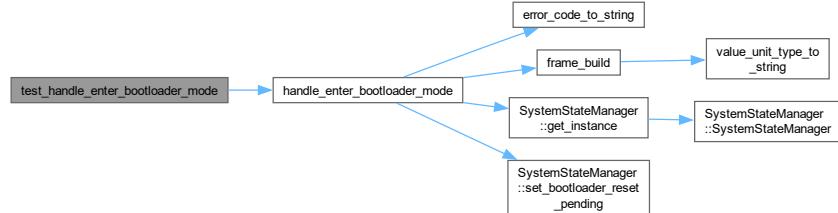


8.123.1.4 `test_handle_enter_bootloader_mode()`

```
void test_handle_enter_bootloader_mode ( void )
```

Definition at line 40 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.124 `test_diagnostic_commands.cpp`

[Go to the documentation of this file.](#)

```
00001 #include "unity.h"
00002 #include "commands.h"
00003 #include "protocol.h"
00004 #include "build_number.h"
00005
```

```
00006 void test_handle_get_commands_list(void) {
00007     std::vector<Frame> response = handle_get_commands_list("", OperationType::GET);
00008
00009     TEST_ASSERT_TRUE(response.size() > 0);
00010     TEST_ASSERT_EQUAL(OperationType::SEQ, response[0].operationType);
00011     TEST_ASSERT_EQUAL(1, response[0].group);
00012     TEST_ASSERT_EQUAL(0, response[0].command);
00013 }
00014
00015 void test_handle_get_build_version(void) {
00016     std::vector<Frame> response = handle_get_build_version("", OperationType::GET);
00017
00018     TEST_ASSERT_EQUAL(1, response.size());
00019     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00020     TEST_ASSERT_EQUAL(1, response[0].group);
00021     TEST_ASSERT_EQUAL(1, response[0].command);
00022     TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
00023 }
00024
00025 void test_handle_verbosity(void) {
00026     // Test SET operation
00027     std::vector<Frame> response = handle_verbosity("2", OperationType::SET);
00028     TEST_ASSERT_EQUAL(1, response.size());
00029     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00030     TEST_ASSERT_EQUAL(1, response[0].group);
00031     TEST_ASSERT_EQUAL(8, response[0].command);
00032     TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());
00033
00034     // Test GET operation
00035     response = handle_verbosity("", OperationType::GET);
00036     TEST_ASSERT_EQUAL(1, response.size());
00037     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00038 }
00039
00040 void test_handle_enter_bootloader_mode(void) {
00041     std::vector<Frame> response = handle_enter_bootloader_mode("", OperationType::SET);
00042
00043     TEST_ASSERT_EQUAL(1, response.size());
00044     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00045     TEST_ASSERT_EQUAL(1, response[0].group);
00046     TEST_ASSERT_EQUAL(9, response[0].command);
00047 }
```

8.125 test/comms/commands/test_event_commands.cpp File Reference

8.126 test_event_commands.cpp

[Go to the documentation of this file.](#)

```
00001
```

8.127 test/comms/commands/test_gps_commands.cpp File Reference

8.128 test_gps_commands.cpp

[Go to the documentation of this file.](#)

```
00001
```

8.129 test/comms/commands/test_power_commands.cpp File Reference

8.130 test_power_commands.cpp

[Go to the documentation of this file.](#)

```
00001
```

8.131 test/comms/commands/test_sensor_commands.cpp File Reference

8.132 test_sensor_commands.cpp

[Go to the documentation of this file.](#)

00001

8.133 test/comms/commands/test_storage_commands.cpp File Reference

8.134 test_storage_commands.cpp

[Go to the documentation of this file.](#)

00001

8.135 test/comms/commands/test_telemetry_commands.cpp File Reference

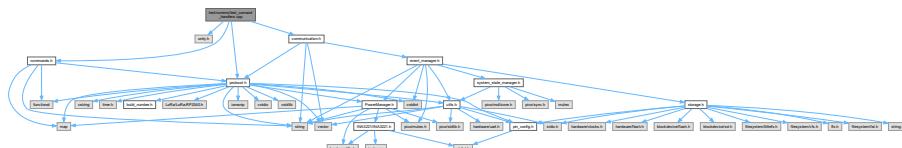
8.136 test_telemetry_commands.cpp

[Go to the documentation of this file.](#)

00001

8.137 test/comms/test_comand_handlers.cpp File Reference

```
#include "unity.h"
#include "protocol.h"
#include "communication.h"
#include "commands.h"
Include dependency graph for test_comand_handlers.cpp:
```



Functions

- void `send_frame_uart` (const `Frame` &`frame`)
- void `send_frame_lora` (const `Frame` &`frame`)
- void `setUp` (void)
- void `tearDown` (void)
- void `test_command_handler_get_operation` (void)
- void `test_command_handler_set_operation` (void)
- void `test_command_handler_invalid_operation` (void)

Variables

- static bool `uart_send_called` = false
- static bool `lora_send_called` = false
- static `Frame` `last_frame_sent`

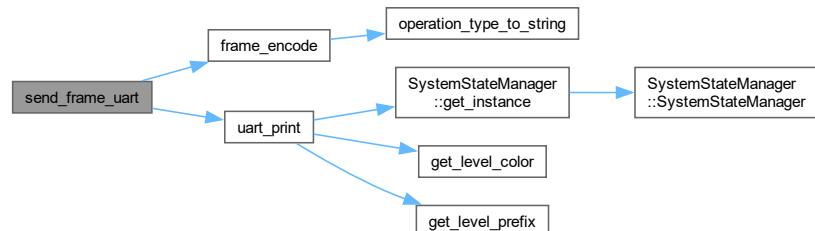
8.137.1 Function Documentation

8.137.1.1 `send_frame_uart()`

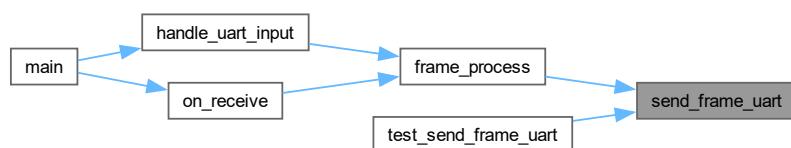
```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 11 of file `test_comand_handlers.cpp`.

Here is the call graph for this function:



Here is the caller graph for this function:

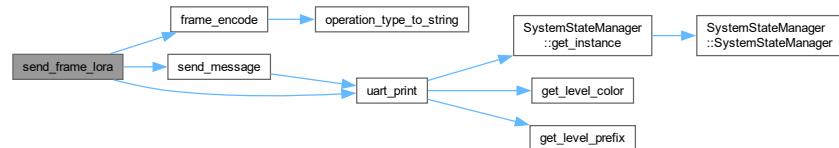


8.137.1.2 send_frame_lora()

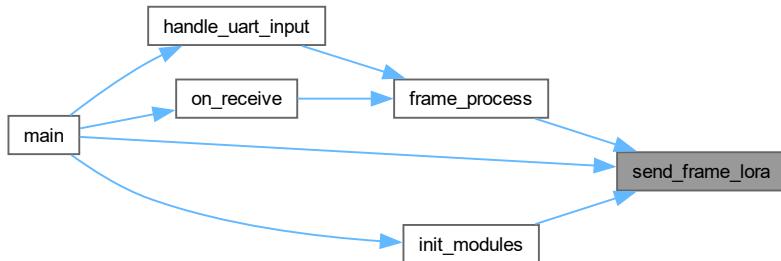
```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 16 of file [test_command_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.137.1.3 setUp()

```
void setUp (
    void )
```

Definition at line 21 of file [test_command_handlers.cpp](#).

8.137.1.4 tearDown()

```
void tearDown (
    void )
```

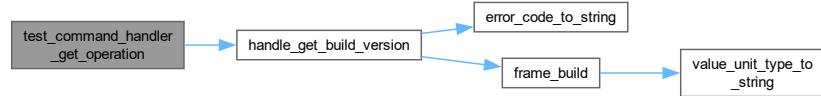
Definition at line 26 of file [test_command_handlers.cpp](#).

8.137.1.5 test_command_handler_get_operation()

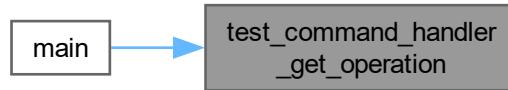
```
void test_command_handler_get_operation (
    void )
```

Definition at line 29 of file [test_command_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

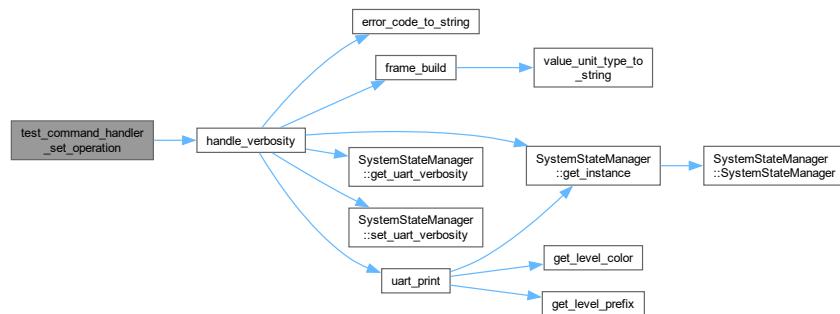


8.137.1.6 test_command_handler_set_operation()

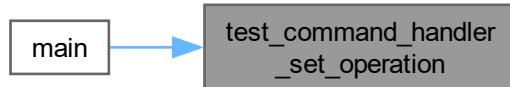
```
void test_command_handler_set_operation (
    void )
```

Definition at line 39 of file [test_command_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

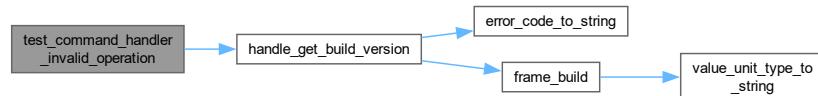


8.137.1.7 `test_command_handler_invalid_operation()`

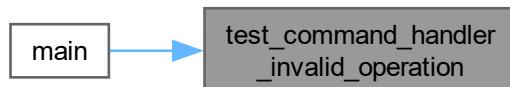
```
void test_command_handler_invalid_operation (
    void )
```

Definition at line 54 of file [test_command_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.137.2 Variable Documentation

8.137.2.1 `uart_send_called`

```
bool uart_send_called = false [static]
```

Definition at line 7 of file [test_command_handlers.cpp](#).

8.137.2.2 lora_send_called

```
bool lora_send_called = false [static]
```

Definition at line 8 of file [test_comand_handlers.cpp](#).

8.137.2.3 last_frame_sent

```
Frame last_frame_sent [static]
```

Definition at line 9 of file [test_comand_handlers.cpp](#).

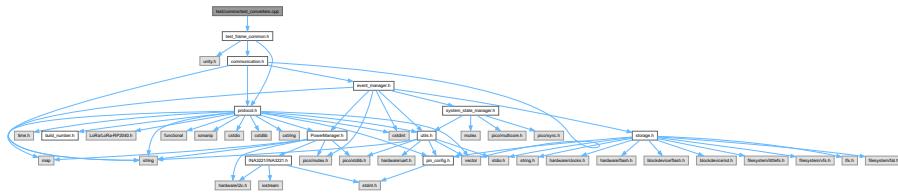
8.138 test_comand_handlers.cpp

[Go to the documentation of this file.](#)

```
00001 // test/comms/test_command_handlers.cpp
00002 #include "unity.h"
00003 #include "protocol.h"
00004 #include "communication.h"
00005 #include "commands.h"
00006
00007 static bool uart_send_called = false;
00008 static bool lora_send_called = false;
00009 static Frame last_frame_sent;
00010
00011 void send_frame_uart(const Frame& frame) {
00012     uart_send_called = true;
00013     last_frame_sent = frame;
00014 }
00015
00016 void send_frame_lora(const Frame& frame) {
00017     lora_send_called = true;
00018     last_frame_sent = frame;
00019 }
00020
00021 void setUp(void) {
00022     uart_send_called = false;
00023     lora_send_called = false;
00024 }
00025
00026 void tearDown(void) {
00027 }
00028
00029 void test_command_handler_get_operation(void) {
00030     std::vector<Frame> response = handle_get_build_version("", OperationType::GET);
00031
00032     TEST_ASSERT_EQUAL(1, response.size());
00033     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00034     TEST_ASSERT_EQUAL(1, response[0].group);
00035     TEST_ASSERT_EQUAL(1, response[0].command);
00036     TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
00037 }
00038
00039 void test_command_handler_set_operation(void) {
00040     VerbosityLevel old_level = get_verbosity_level();
00041     std::vector<Frame> response = handle_verbosity("2", OperationType::SET);
00042
00043     TEST_ASSERT_EQUAL(1, response.size());
00044     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00045     TEST_ASSERT_EQUAL(1, response[0].group);
00046     TEST_ASSERT_EQUAL(8, response[0].command);
00047     TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());
00048
00049     TEST_ASSERT_EQUAL(VerbosityLevel::WARNING, get_verbosity_level());
00050
00051     set_verbosity_level(old_level);
00052 }
00053
00054 void test_command_handler_invalid_operation(void) {
00055     std::vector<Frame> response = handle_get_build_version("", OperationType::SET);
00056
00057     TEST_ASSERT_EQUAL(1, response.size());
00058     TEST_ASSERT_EQUAL(OperationType::ERR, response[0].operationType);
00059     TEST_ASSERT_EQUAL(1, response[0].group);
00060     TEST_ASSERT_EQUAL(1, response[0].command);
00061 }
```

8.139 test/comms/test_converters.cpp File Reference

```
#include "test_frame_common.h"
Include dependency graph for test_converters.cpp:
```



Functions

- void [test_operation_type_conversion \(\)](#)
- void [test_value_unit_type_conversion \(\)](#)
- void [test_exception_type_conversion \(\)](#)
- void [test_hex_string_conversion \(\)](#)

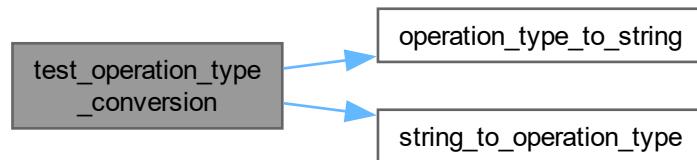
8.139.1 Function Documentation

8.139.1.1 [test_operation_type_conversion\(\)](#)

```
void test_operation_type_conversion (
    void )
```

Definition at line 4 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

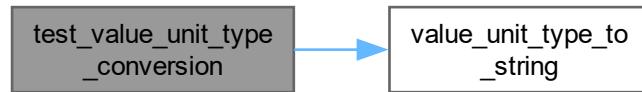


8.139.1.2 test_value_unit_type_conversion()

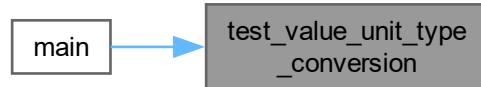
```
void test_value_unit_type_conversion (
    void )
```

Definition at line 13 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

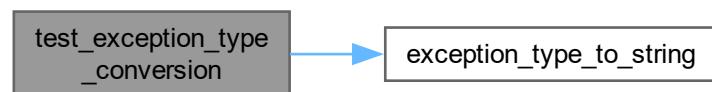


8.139.1.3 test_exception_type_conversion()

```
void test_exception_type_conversion (
    void )
```

Definition at line 20 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.139.1.4 `test_hex_string_conversion()`

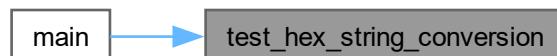
```
void test_hex_string_conversion (
    void )
```

Definition at line 27 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.140 `test_converters.cpp`

[Go to the documentation of this file.](#)

```
00001 // test_frame_converters.cpp
00002 #include "test_frame_common.h"
00003
00004 void test_operation_type_conversion() {
00005     OperationType type = OperationType::GET;
00006     std::string str = operation_type_to_string(type);
```

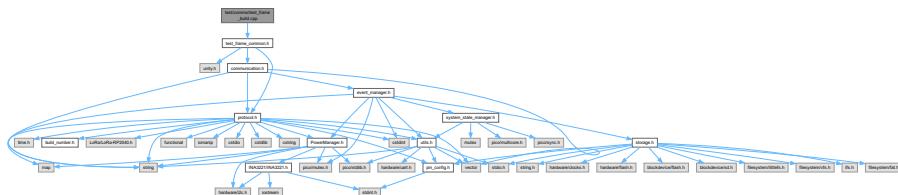
```

00007     OperationType converted = string_to_operation_type(str);
00008
00009     TEST_ASSERT_EQUAL(type, converted);
00010     TEST_ASSERT_EQUAL_STRING("GET", str.c_str());
00011 }
00012
0013 void test_value_unit_type_conversion() {
0014     ValueUnit unit = ValueUnit::VOLT;
0015     std::string str = value_unit_type_to_string(unit);
0016
0017     TEST_ASSERT_EQUAL_STRING("V", str.c_str());
0018 }
0019
0020 void test_exception_type_conversion() {
0021     ExceptionType type = ExceptionType::INVALID_PARAM;
0022     std::string str = exception_type_to_string(type);
0023
0024     TEST_ASSERT_EQUAL_STRING("INVALID PARAM", str.c_str());
0025 }
0026
0027 void test_hex_string_conversion() {
0028     std::string hex = "0A0B0C";
0029     std::vector<uint8_t> bytes = hex_string_to_bytes(hex);
0030
0031     TEST_ASSERT_EQUAL(3, bytes.size());
0032     TEST_ASSERT_EQUAL(0x0A, bytes[0]);
0033     TEST_ASSERT_EQUAL(0x0B, bytes[1]);
0034     TEST_ASSERT_EQUAL(0x0C, bytes[2]);
0035 }

```

8.141 test/comms/test_frame_build.cpp File Reference

#include "test_frame_common.h"
Include dependency graph for test_frame_build.cpp:



Functions

- void [test_frame_build_val\(\)](#)
- void [test_frame_build_err\(\)](#)
- void [test_frame_build_get\(\)](#)
- void [test_frame_build_set\(\)](#)
- void [test_frame_build_res\(\)](#)
- void [test_frame_build_seq\(\)](#)

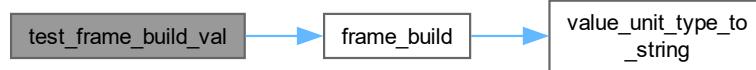
8.141.1 Function Documentation

8.141.1.1 [test_frame_build_val\(\)](#)

```
void test_frame_build_val (
    void )
```

Definition at line 4 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



8.141.1.2 test_frame_build_err()

```
void test_frame_build_err (
    void )
```

Definition at line 15 of file [test_frame_build.cpp](#).

Here is the call graph for this function:

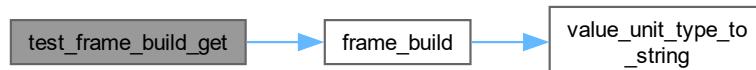


8.141.1.3 test_frame_build_get()

```
void test_frame_build_get (
    void )
```

Definition at line 24 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.141.1.4 test_frame_build_set()

```
void test_frame_build_set (
    void )
```

Definition at line 35 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.141.1.5 test_frame_build_res()

```
void test_frame_build_res (
    void )
```

Definition at line 46 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.141.1.6 `test_frame_build_seq()`

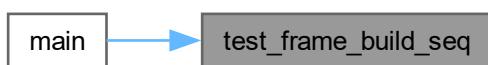
```
void test_frame_build_seq ( void )
```

Definition at line 57 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



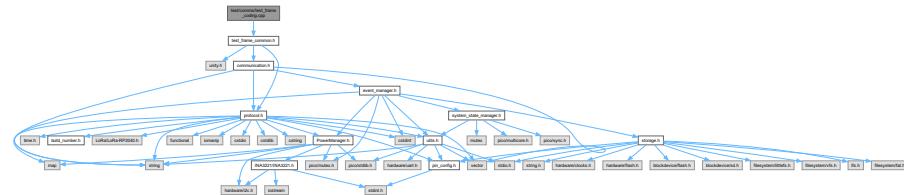
8.142 test_frame_build.cpp

[Go to the documentation of this file.](#)

```
00001 // test_frame_build.cpp
00002 #include "test_frame_common.h"
00003
00004 void test_frame_build_val() {
00005     Frame frame = frame_build(OperationType::VAL, 1, 2, "test_value", ValueUnit::VOLT);
00006
00007     TEST_ASSERT_EQUAL(1, frame.direction);
00008     TEST_ASSERT_EQUAL(OperationType::VAL, frame.operationType);
00009     TEST_ASSERT_EQUAL(1, frame.group);
00010     TEST_ASSERT_EQUAL(2, frame.command);
00011     TEST_ASSERT_EQUAL_STRING("test_value", frame.value.c_str());
00012     TEST_ASSERT_EQUAL_STRING("V", frame.unit.c_str());
00013 }
00014
00015 void test_frame_build_err() {
00016     Frame frame = frame_build(OperationType::ERR, 1, 2, "error_message");
00017
00018     TEST_ASSERT_EQUAL(1, frame.direction);
00019     TEST_ASSERT_EQUAL(OperationType::ERR, frame.operationType);
00020     TEST_ASSERT_EQUAL_STRING("error_message", frame.value.c_str());
00021     TEST_ASSERT_EQUAL_STRING("", frame.unit.c_str());
00022 }
00023
00024 void test_frame_build_get() {
00025     Frame frame = frame_build(OperationType::GET, 3, 4, "");
00026
00027     TEST_ASSERT_EQUAL(0, frame.direction); // Ground to satellite
00028     TEST_ASSERT_EQUAL(OperationType::GET, frame.operationType);
00029     TEST_ASSERT_EQUAL(3, frame.group);
00030     TEST_ASSERT_EQUAL(4, frame.command);
00031     TEST_ASSERT_EQUAL_STRING("", frame.value.c_str());
00032     TEST_ASSERT_EQUAL_STRING("", frame.unit.c_str());
00033 }
00034
00035 void test_frame_build_set() {
00036     Frame frame = frame_build(OperationType::SET, 5, 6, "set_value", ValueUnit::SECOND);
00037
00038     TEST_ASSERT_EQUAL(0, frame.direction); // Ground to satellite
00039     TEST_ASSERT_EQUAL(OperationType::SET, frame.operationType);
00040     TEST_ASSERT_EQUAL(5, frame.group);
00041     TEST_ASSERT_EQUAL(6, frame.command);
00042     TEST_ASSERT_EQUAL_STRING("set_value", frame.value.c_str());
00043     TEST_ASSERT_EQUAL_STRING("s", frame.unit.c_str()); // Assuming SECOND is converted to "s"
00044 }
00045
00046 void test_frame_build_res() {
00047     Frame frame = frame_build(OperationType::RES, 7, 8, "result", ValueUnit::BOOL);
00048
00049     TEST_ASSERT_EQUAL(1, frame.direction); // Satellite to ground
00050     TEST_ASSERT_EQUAL(OperationType::RES, frame.operationType);
00051     TEST_ASSERT_EQUAL(7, frame.group);
00052     TEST_ASSERT_EQUAL(8, frame.command);
00053     TEST_ASSERT_EQUAL_STRING("result", frame.value.c_str());
00054     TEST_ASSERT_EQUAL_STRING("bool", frame.unit.c_str()); // Assuming BOOL is converted to "bool"
00055 }
00056
00057 void test_frame_build_seq() {
00058     Frame frame = frame_build(OperationType::SEQ, 9, 10, "sequence_data", ValueUnit::TEXT);
00059
00060     TEST_ASSERT_EQUAL(1, frame.direction); // Satellite to ground
00061     TEST_ASSERT_EQUAL(OperationType::SEQ, frame.operationType);
00062     TEST_ASSERT_EQUAL(9, frame.group);
00063     TEST_ASSERT_EQUAL(10, frame.command);
00064     TEST_ASSERT_EQUAL_STRING("sequence_data", frame.value.c_str());
00065     TEST_ASSERT_EQUAL_STRING("text", frame.unit.c_str()); // Assuming TEXT is converted to "text"
00066 }
```

8.143 test/comms/test_frame_coding.cpp File Reference

```
#include "test_frame_common.h"
Include dependency graph for test_frame_coding.cpp:
```



Functions

- void [test_frame_encode_basic \(\)](#)
- void [test_frame_decode_basic \(\)](#)
- void [test_frame_decode_invalid_header \(\)](#)

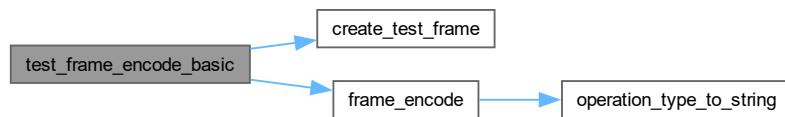
8.143.1 Function Documentation

8.143.1.1 [test_frame_encode_basic\(\)](#)

```
void test_frame_encode_basic (
    void )
```

Definition at line 4 of file [test_frame_coding.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

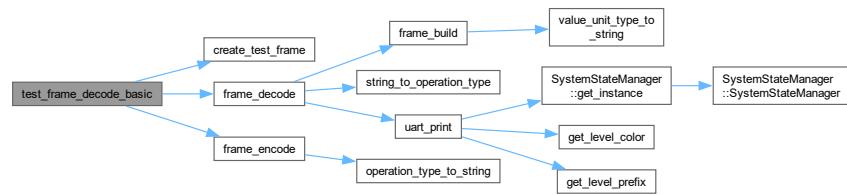


8.143.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
    void )
```

Definition at line 14 of file [test_frame_coding.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

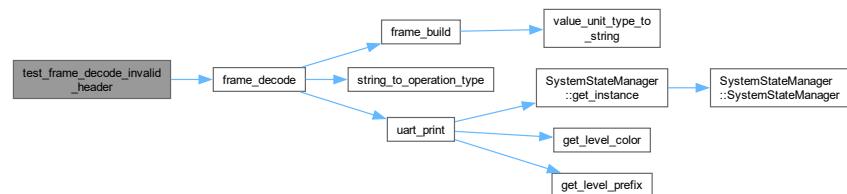


8.143.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
    void )
```

Definition at line 26 of file [test_frame_coding.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.144 test_frame_coding.cpp

[Go to the documentation of this file.](#)

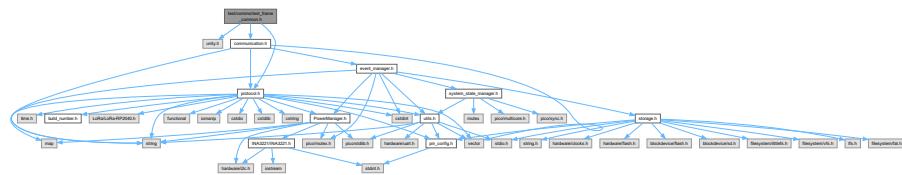
```

00001 // test_frame_codec.cpp
00002 #include "test_frame_common.h"
00003
00004 void test_frame_encode_basic() {
00005     Frame frame = create_test_frame();
00006     std::string encoded = frame_encode(frame);
00007
00008     TEST_ASSERT_NOT_EQUAL(0, encoded.length());
00009     TEST_ASSERT_TRUE(encoded.find(FRAME_BEGIN) != std::string::npos);
00010     TEST_ASSERT_TRUE(encoded.find(FRAME_END) != std::string::npos);
00011     TEST_ASSERT_TRUE(encoded.find("test_value") != std::string::npos);
00012 }
00013
00014 void test_frame_decode_basic() {
00015     Frame original = create_test_frame();
00016     std::string encoded = frame_encode(original);
00017     Frame decoded = frame_decode(encoded);
00018
00019     TEST_ASSERT_EQUAL(original.direction, decoded.direction);
00020     TEST_ASSERT_EQUAL(original.group, decoded.group);
00021     TEST_ASSERT_EQUAL(original.command, decoded.command);
00022     TEST_ASSERT_EQUAL_STRING(original.value.c_str(), decoded.value.c_str());
00023     TEST_ASSERT_EQUAL_STRING(original.unit.c_str(), decoded.unit.c_str());
00024 }
00025
00026 void test_frame_decode_invalid_header() {
00027     std::string invalid_frame = "INVALID" + std::string(1, DELIMITER) + "rest_of_frame";
00028     bool exceptionThrown = false;
00029
00030     try {
00031         Frame decoded = frame_decode(invalid_frame);
00032     } catch (const std::runtime_error& e) {
00033         exceptionThrown = true;
00034     } catch (...) {
00035         // Catch any other exceptions to avoid crashing the test
00036     }
00037
00038     TEST_ASSERT_TRUE(exceptionThrown);
00039 }
```

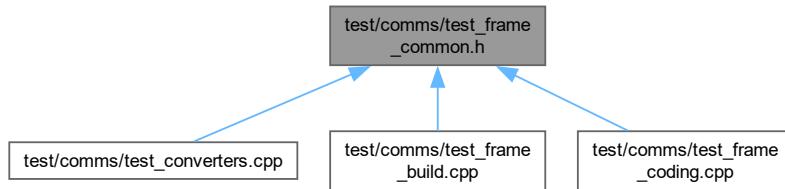
8.145 test/comms/test_frame_common.h File Reference

```
#include "unity.h"
#include "protocol.h"
```

```
#include "communication.h"
Include dependency graph for test_frame_common.h:
```



This graph shows which files directly or indirectly include this file:



Functions

- Frame [create_test_frame \(\)](#)

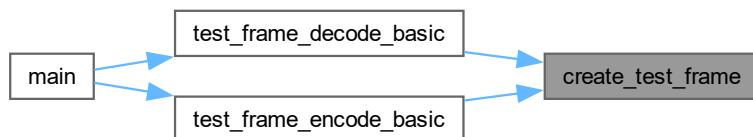
8.145.1 Function Documentation

8.145.1.1 [create_test_frame\(\)](#)

`Frame create_test_frame () [inline]`

Definition at line 9 of file [test_frame_common.h](#).

Here is the caller graph for this function:



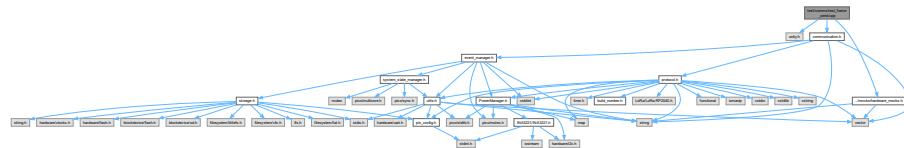
8.146 test_frame_common.h

[Go to the documentation of this file.](#)

```
00001 // test_frame_common.h
00002 #ifndef TEST_FRAME_COMMON_H
00003 #define TEST_FRAME_COMMON_H
00004
00005 #include "unity.h"
00006 #include "protocol.h"
00007 #include "communication.h"
00008
00009 inline Frame create_test_frame() {
00010     Frame frame;
00011     frame.header = FRAME_BEGIN;
00012     frame.direction = 1;
00013     frame.operationType = OperationType::GET;
00014     frame.group = 1;
00015     frame.command = 2;
00016     frame.value = "test_value";
00017     frame.unit = "V";
00018     frame.footer = FRAME_END;
00019     return frame;
00020 }
00021
00022 #endif
```

8.147 test/comms/test_frame_send.cpp File Reference

```
#include "unity.h"
#include "communication.h"
#include "../mocks/hardware_mock.h"
Include dependency graph for test_frame_send.cpp:
```



Functions

- void [setUp](#) (void)
- void [tearDown](#) (void)
- void [test_send_frame_uart](#) (void)

8.147.1 Function Documentation

8.147.1.1 [setUp\(\)](#)

```
void setUp (
    void )
```

Definition at line 6 of file [test_frame_send.cpp](#).

8.147.1.2 tearDown()

```
void tearDown (
    void )
```

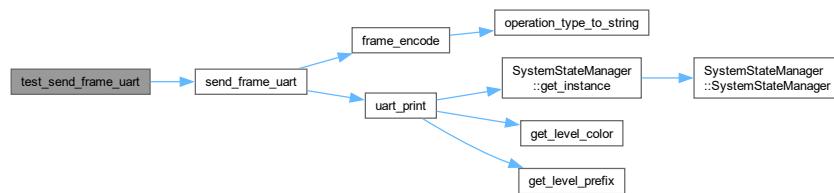
Definition at line 12 of file [test_frame_send.cpp](#).

8.147.1.3 test_send_frame_uart()

```
void test_send_frame_uart (
    void )
```

Definition at line 17 of file [test_frame_send.cpp](#).

Here is the call graph for this function:



8.148 test_frame_send.cpp

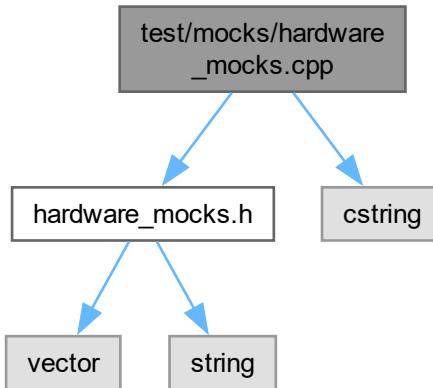
[Go to the documentation of this file.](#)

```

00001 // test/comms/test_frame_send.cpp
00002 #include "unity.h"
00003 #include "communication.h"
00004 #include "../mocks/hardware_mock.h"
00005
00006 void setUp(void) {
00007     // Enable mocks before each test
00008     mock_uart_enabled = true;
00009     uart_output_buffer.clear();
00010 }
00011
00012 void tearDown(void) {
00013     // Disable mocks after each test
00014     mock_uart_enabled = false;
00015 }
00016
00017 void test_send_frame_uart(void) {
00018     // Create a test frame
00019     Frame test_frame = {
00020         .operationType = OperationType::VAL,
00021         .group = 1,
00022         .command = 2,
00023         .value = "TEST_VALUE"
00024     };
00025
00026     // Call function under test
00027     send_frame_uart(test_frame);
00028
00029     // Verify output using mocks
00030     TEST_ASSERT_EQUAL(1, uart_output_buffer.size());
00031     TEST_ASSERT_TRUE(uart_output_buffer[0].find("KBST;0;VAL;1;2;TEST_VALUE;") != std::string::npos);
00032 }
```

8.149 test/mock/hardware_mocks.cpp File Reference

```
#include "hardware_mocks.h"
#include <cstring>
Include dependency graph for hardware_mocks.cpp:
```



Functions

- void [mock_uart_puts](#) (uart_inst_t *uart, const char *str)
- void [mock_uart_init](#) (uart_inst_t *uart, uint baudrate)
- void [mock_spi_write_blocking](#) (spi_inst_t *spi, const uint8_t *src, size_t len)
- int [mock_spi_read_blocking](#) (spi_inst_t *spi, uint8_t tx_data, uint8_t *dst, size_t len)

Variables

- bool [mock_uart_enabled](#) = false
- std::vector< std::string > [uart_output_buffer](#)
- bool [mock_spi_enabled](#) = false
- std::vector< uint8_t > [spi_output_buffer](#)

8.149.1 Function Documentation

8.149.1.1 [mock_uart_puts\(\)](#)

```
void mock_uart_puts (
    uart_inst_t * uart,
    const char * str)
```

Definition at line 9 of file [hardware_mocks.cpp](#).

8.149.1.2 mock_uart_init()

```
void mock_uart_init (
    uart_inst_t * uart,
    uint baudrate)
```

Definition at line 17 of file [hardware_mocks.cpp](#).

8.149.1.3 mock_spi_write_blocking()

```
void mock_spi_write_blocking (
    spi_inst_t * spi,
    const uint8_t * src,
    size_t len)
```

Definition at line 27 of file [hardware_mocks.cpp](#).

8.149.1.4 mock_spi_read_blocking()

```
int mock_spi_read_blocking (
    spi_inst_t * spi,
    uint8_t tx_data,
    uint8_t * dst,
    size_t len)
```

Definition at line 35 of file [hardware_mocks.cpp](#).

8.149.2 Variable Documentation

8.149.2.1 mock_uart_enabled

```
bool mock_uart_enabled = false
```

Definition at line 6 of file [hardware_mocks.cpp](#).

8.149.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer
```

Definition at line 7 of file [hardware_mocks.cpp](#).

8.149.2.3 mock_spi_enabled

```
bool mock_spi_enabled = false
```

Definition at line 24 of file [hardware_mocks.cpp](#).

8.149.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer
```

Definition at line 25 of file [hardware_mocks.cpp](#).

8.150 hardware_mocks.cpp

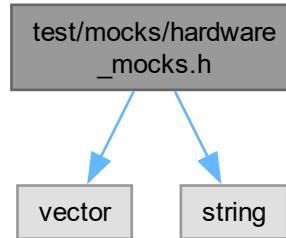
[Go to the documentation of this file.](#)

```
00001 // test/mock/hardwareMocks.cpp
00002 #include "hardwareMocks.h"
00003 #include <cstring>
00004
00005 // UART mocks
00006 bool mock_uart_enabled = false;
00007 std::vector<std::string> uart_output_buffer;
00008
00009 void mock_uart_puts(uart_inst_t* uart, const char* str) {
00010     if (mock_uart_enabled) {
00011         uart_output_buffer.push_back(std::string(str));
00012     } else {
00013         uart_puts(uart, str);
00014     }
00015 }
00016
00017 void mock_uart_init(uart_inst_t* uart, uint baudrate) {
00018     if (!mock_uart_enabled) {
00019         uart_init(uart, baudrate);
00020     }
00021 }
00022
00023 // SPI mocks
00024 bool mock_spi_enabled = false;
00025 std::vector<uint8_t> spi_output_buffer;
00026
00027 void mock_spi_write_blocking(spi_inst_t* spi, const uint8_t* src, size_t len) {
00028     if (mock_spi_enabled) {
00029         spi_output_buffer.insert(spi_output_buffer.end(), src, src + len);
00030     } else {
00031         spi_write_blocking(spi, src, len);
00032     }
00033 }
00034
00035 int mock_spi_read_blocking(spi_inst_t* spi, uint8_t tx_data, uint8_t* dst, size_t len) {
00036     if (mock_spi_enabled) {
00037         // Mock implementation that fills dst with test data
00038         memset(dst, tx_data, len);
00039         return len;
00040     } else {
00041         return spi_read_blocking(spi, tx_data, dst, len);
00042     }
00043 }
```

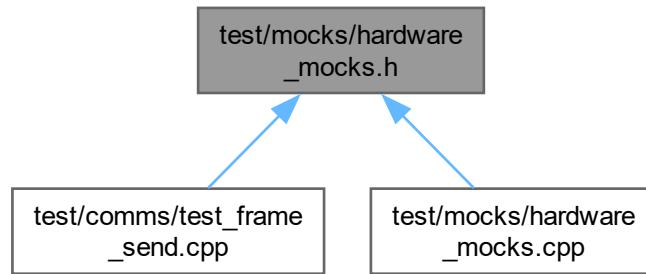
8.151 test/mock/hardwareMocks.h File Reference

```
#include <vector>
#include <string>
```

Include dependency graph for hardware_mocksh:



This graph shows which files directly or indirectly include this file:



Functions

- void [mock_uart_puts](#) (uart_inst_t *uart, const char *str)
- void [mock_uart_init](#) (uart_inst_t *uart, uint baudrate)
- void [mock_spi_write_blocking](#) (spi_inst_t *spi, const uint8_t *src, size_t len)
- int [mock_spi_read_blocking](#) (spi_inst_t *spi, uint8_t tx_data, uint8_t *dst, size_t len)

Variables

- bool [mock_uart_enabled](#)
- std::vector< std::string > [uart_output_buffer](#)
- bool [mock_spi_enabled](#)
- std::vector< uint8_t > [spi_output_buffer](#)

8.151.1 Function Documentation

8.151.1.1 mock_uart_puts()

```
void mock_uart_puts (
    uart_inst_t * uart,
    const char * str)
```

Definition at line 9 of file [hardwareMocks.cpp](#).

8.151.1.2 mock_uart_init()

```
void mock_uart_init (
    uart_inst_t * uart,
    uint baudrate)
```

Definition at line 17 of file [hardwareMocks.cpp](#).

8.151.1.3 mock_spi_write_blocking()

```
void mock_spi_write_blocking (
    spi_inst_t * spi,
    const uint8_t * src,
    size_t len)
```

Definition at line 27 of file [hardwareMocks.cpp](#).

8.151.1.4 mock_spi_read_blocking()

```
int mock_spi_read_blocking (
    spi_inst_t * spi,
    uint8_t tx_data,
    uint8_t * dst,
    size_t len)
```

Definition at line 35 of file [hardwareMocks.cpp](#).

8.151.2 Variable Documentation

8.151.2.1 mock_uart_enabled

```
bool mock_uart_enabled [extern]
```

Definition at line 6 of file [hardwareMocks.cpp](#).

8.151.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer [extern]
```

Definition at line 7 of file [hardwareMocks.cpp](#).

8.151.2.3 mock_spi_enabled

```
bool mock_spi_enabled [extern]
```

Definition at line 24 of file [hardwareMocks.cpp](#).

8.151.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer [extern]
```

Definition at line 25 of file [hardwareMocks.cpp](#).

8.152 hardware_mocks.h

[Go to the documentation of this file.](#)

```
00001 // test/mock/hardwareMocks.h
00002 #ifndef HARDWARE_MOCKS_H
00003 #define HARDWARE_MOCKS_H
00004
00005 #include <vector>
00006 #include <string>
00007
00008 // UART mocks
00009 extern bool mock_uart_enabled;
00010 extern std::vector<std::string> uart_output_buffer;
00011
00012 void mock_uart_puts(uart_inst_t* uart, const char* str);
00013 void mock_uart_init(uart_inst_t* uart, uint baudrate);
00014
00015 // SPI mocks
00016 extern bool mock_spi_enabled;
00017 extern std::vector<uint8_t> spi_output_buffer;
00018
00019 void mock_spi_write_blocking(spi_inst_t* spi, const uint8_t* src, size_t len);
00020 int mock_spi_read_blocking(spi_inst_t* spi, uint8_t* tx_data, uint8_t* dst, size_t len);
00021
00022 #endif // HARDWARE_MOCKS_H
```

8.153 test/test_mocks.cpp File Reference

```
#include "utils.h"
#include "protocol.h"
#include "PowerManager.h"
#include "DS3231.h"
#include "lib/location/NMEA/NMEA_data.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <map>
#include <cstdio>
```

Include dependency graph for `test_mocks.cpp`:



Functions

- void `uart_print` (const std::string &msg, VerbosityLevel level, bool add_timestamp, uart_inst_t *uart)
- std::string `get_last_telemetry_record_csv` ()

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.
- void `test_error_code_conversion` ()
- void `test_command_handler_get_operation` ()
- void `test_command_handler_set_operation` ()
- void `test_command_handler_invalid_operation` ()

Variables

- PowerManager `powerManager` (nullptr)
- DS3231 `systemClock` (nullptr)
- NMEAData `nmea_data`
- volatile bool `g_pending_bootloader_reset` = false
- VerbosityLevel `g_uart_verbosity` = VerbosityLevel::DEBUG
- std::string `uart_output_buffer`
- bool `mock_uart_enabled` = false

8.153.1 Function Documentation

8.153.1.1 `uart_print()`

```
void uart_print (
    const std::string & msg,
    VerbosityLevel level,
    bool add_timestamp,
    uart_inst_t * uart)
```

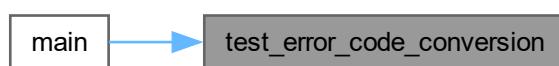
Definition at line 26 of file [test_mocks.cpp](#).

8.153.1.2 `test_error_code_conversion()`

```
void test_error_code_conversion (
    void )
```

Definition at line 101 of file [test_mocks.cpp](#).

Here is the caller graph for this function:



8.153.1.3 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
    void )
```

Definition at line 102 of file [test_mocks.cpp](#).

8.153.1.4 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
    void )
```

Definition at line 103 of file [test_mocks.cpp](#).

8.153.1.5 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
    void )
```

Definition at line 104 of file [test_mocks.cpp](#).

8.153.2 Variable Documentation

8.153.2.1 powerManager

```
PowerManager powerManager(nullptr) (
    nullptr )
```

8.153.2.2 systemClock

```
DS3231 systemClock(nullptr) (
    nullptr )
```

8.153.2.3 nmea_data

```
NMEAData nmea_data
```

Definition at line 17 of file [test_mocks.cpp](#).

8.153.2.4 g_pending_bootloader_reset

```
volatile bool g_pending_bootloader_reset = false
```

Definition at line 18 of file [test_mocks.cpp](#).

8.153.2.5 g_uart_verbosity

```
VerbosityLevel g_uart_verbosity = VerbosityLevel::DEBUG
```

Definition at line 19 of file [test_mocks.cpp](#).

8.153.2.6 uart_output_buffer

```
std::string uart_output_buffer
```

Definition at line 22 of file [test_mocks.cpp](#).

8.153.2.7 mock_uart_enabled

```
bool mock_uart_enabled = false
```

Definition at line 23 of file [test_mocks.cpp](#).

8.154 test_mocks.cpp

[Go to the documentation of this file.](#)

```
00001 // test/test_mocks.cpp - Mock implementations for unit tests
00002
00003 #include "utils.h"
00004 #include "protocol.h"
00005 #include "PowerManager.h"
00006 #include "DS3231.h"
00007 #include "lib/location/NMEA/NMEA_data.h"
00008 #include "ISensor.h"
00009 #include <vector>
00010 #include <string>
00011 #include <map>
00012 #include <cstdio>
00013
00014 // Mock global variables
00015 PowerManager powerManager(nullptr);
00016 DS3231 systemClock(nullptr);
00017 NMEAData nmea_data;
00018 volatile bool g_pending_bootloader_reset = false;
00019 VerbosityLevel g_uart_verbosity = VerbosityLevel::DEBUG;
00020
00021 // Mock UART buffers for testing
00022 std::string uart_output_buffer;
00023 bool mock_uart_enabled = false;
00024
00025 // Mock UART print function
00026 void uart_print(const std::string& msg, VerbosityLevel level, bool add_timestamp, uart_inst_t* uart) {
00027     if (mock_uart_enabled) {
00028         uart_output_buffer += msg + "\n";
00029     }
00030     // In test environment, we don't actually print to UART
00031 }
00032
00033 // NMEAData methods
00034 NMEAData::NMEAData() {}
00035
00036 std::vector<std::string> NMEAData::get_rmc_tokens() const {
00037     return {"$GPRMC", "123519", "A", "4807.038", "N", "01131.000", "E", "022.4", "084.4", "230394",
00038     "003.1", "W"};
00039
00040 std::vector<std::string> NMEAData::get_gga_tokens() const {
00041     return {"$GPGGA", "123519", "4807.038", "N", "01131.000", "E", "1", "08", "0.9", "545.4", "M",
00042     "46.9", "M", "", ""};
00043
00044 bool NMEAData::has_valid_time() const {
```

```

00045     return true;
00046 }
00047
00048 time_t NMEAData::get_unix_time() const {
00049     return 1234567890; // Some fixed timestamp for tests
00050 }
00051
00052 // Implementation of SensorWrapper singleton methods - not redefining the class itself
00053 // This creates concrete implementations of the methods declared in ISensor.h
00054 SensorWrapper& SensorWrapper::get_instance() {
00055     static SensorWrapper instance;
00056     return instance;
00057 }
00058
00059 bool SensorWrapper::sensor_init(SensorType type, i2c_inst_t* i2c) {
00060     return true; // Always succeed in tests
00061 }
00062
00063 bool SensorWrapper::sensor_configure(SensorType type, const std::map<std::string, std::string>&
00064 config) {
00065     return true; // Always succeed in tests
00066 }
00067 float SensorWrapper::sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType) {
00068     return 42.0f; // Return fixed test value
00069 }
00070
00071 ISensor* SensorWrapper::get_sensor(SensorType type) {
00072     return nullptr; // For tests, we don't need to return actual sensor instances
00073 }
00074
00075 std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::scan_connected_sensors(i2c_inst_t* i2c) {
00076     return {
00077         {SensorType::ENVIRONMENT, 0x76},
00078         {SensorType::LIGHT, 0x23}
00079     };
00080 }
00081
00082 std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::get_available_sensors() {
00083     return {
00084         {SensorType::ENVIRONMENT, 0x76},
00085         {SensorType::LIGHT, 0x23}
00086     };
00087 }
00088
00089 std::string get_last_telemetry_record_csv() {
00090     return "mock_timestamp,mock_value1,mock_value2";
00091 }
00092
00093 std::string get_last_sensor_record_csv() {
00094     return "mock_timestamp,mock_sensor1,mock_sensor2";
00095 }
00096
00097 // Private constructor implementation (that would be defined in ISensor.cpp)
00098 SensorWrapper::SensorWrapper() {}
00099
00100 // Mock test cases that were referenced but not defined
00101 void test_error_code_conversion() {}
00102 void test_command_handler_get_operation() {}
00103 void test_command_handler_set_operation() {}
00104 void test_command_handler_invalid_operation() {}

```

8.155 test/test_runner.cpp File Reference

```
#include "includes.h"
#include "unity.h"
Include dependency graph for test_runner.cpp:
```



Functions

- void `test_frame_encode_basic` (void)
- void `test_frame_decode_basic` (void)
- void `test_frame_decode_invalid_header` (void)
- void `test_frame_build_get` (void)
- void `test_frame_build_set` (void)
- void `test_frame_build_res` (void)
- void `test_frame_build_seq` (void)
- void `test_frame_build_val` (void)
- void `test_frame_build_err` (void)
- void `test_operation_type_conversion` (void)
- void `test_value_unit_type_conversion` (void)
- void `test_exception_type_conversion` (void)
- void `test_hex_string_conversion` (void)
- void `test_command_handler_get_operation` (void)
- void `test_command_handler_set_operation` (void)
- void `test_command_handler_invalid_operation` (void)
- void `test_handle_get_commands_list` (void)
- void `test_handle_get_build_version` (void)
- void `test_handle_verbosity` (void)
- void `test_handle_enter_bootloader_mode` (void)
- void `test_error_code_conversion` (void)
- int `main` (void)

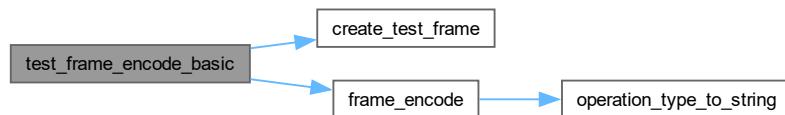
8.155.1 Function Documentation

8.155.1.1 `test_frame_encode_basic()`

```
void test_frame_encode_basic (
    void ) [extern]
```

Definition at line 4 of file `test_frame_coding.cpp`.

Here is the call graph for this function:



Here is the caller graph for this function:

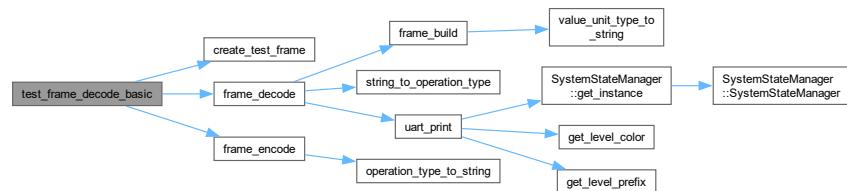


8.155.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
    void ) [extern]
```

Definition at line 14 of file [test_frame_coding.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

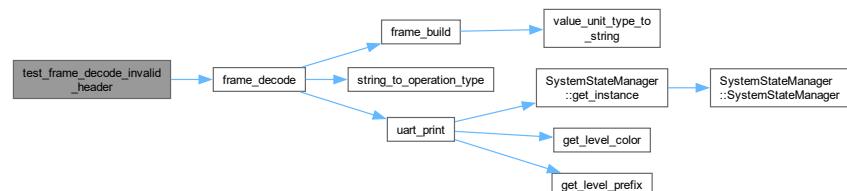


8.155.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
    void ) [extern]
```

Definition at line 26 of file [test_frame_coding.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.4 `test_frame_build_get()`

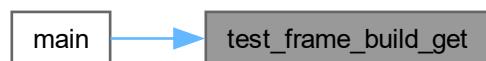
```
void test_frame_build_get (
    void ) [extern]
```

Definition at line 24 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

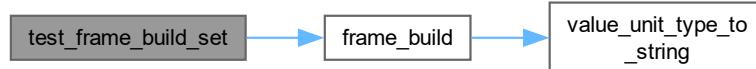


8.155.1.5 `test_frame_build_set()`

```
void test_frame_build_set (
    void ) [extern]
```

Definition at line 35 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.6 test_frame_build_res()

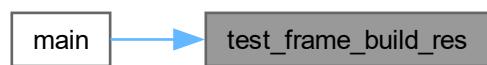
```
void test_frame_build_res (
    void ) [extern]
```

Definition at line 46 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.7 test_frame_build_seq()

```
void test_frame_build_seq (
    void ) [extern]
```

Definition at line 57 of file [test_frame_build.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.8 test_frame_build_val()

```
void test_frame_build_val (
    void ) [extern]
```

Definition at line 4 of file [test_frame_build.cpp](#).

Here is the call graph for this function:

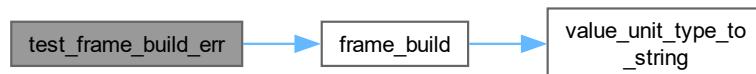


8.155.1.9 test_frame_build_err()

```
void test_frame_build_err (
    void ) [extern]
```

Definition at line 15 of file [test_frame_build.cpp](#).

Here is the call graph for this function:

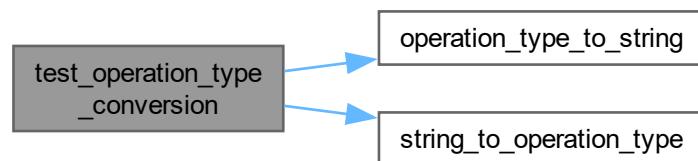


8.155.1.10 test_operation_type_conversion()

```
void test_operation_type_conversion (
    void ) [extern]
```

Definition at line 4 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

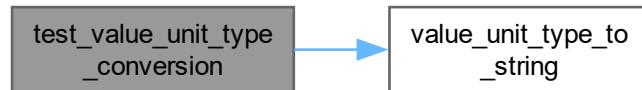


8.155.1.11 test_value_unit_type_conversion()

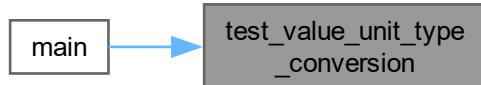
```
void test_value_unit_type_conversion (
    void )  [extern]
```

Definition at line 13 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

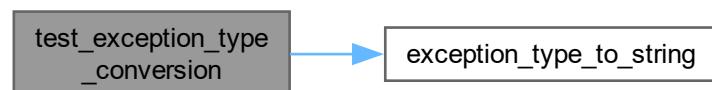


8.155.1.12 test_exception_type_conversion()

```
void test_exception_type_conversion (
    void )  [extern]
```

Definition at line 20 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.13 test_hex_string_conversion()

```
void test_hex_string_conversion (
    void ) [extern]
```

Definition at line 27 of file [test_converters.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

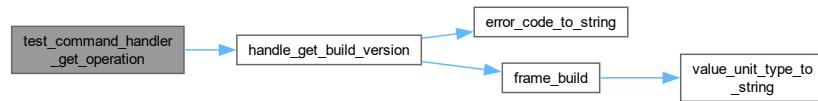


8.155.1.14 test_command_handler_get_operation()

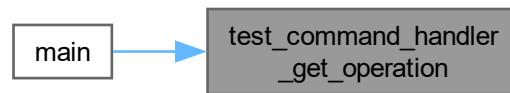
```
void test_command_handler_get_operation (
    void ) [extern]
```

Definition at line 29 of file [test_comand_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

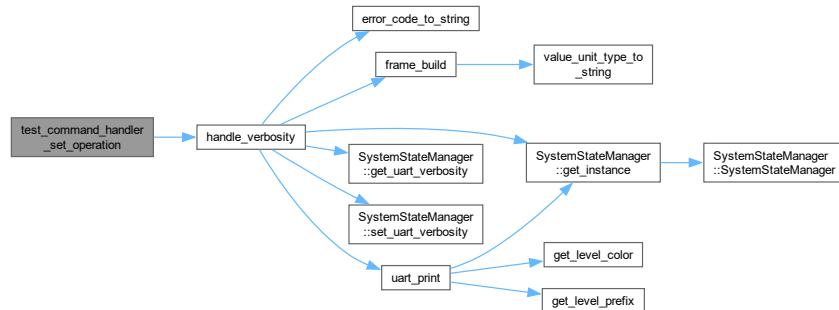


8.155.1.15 test_command_handler_set_operation()

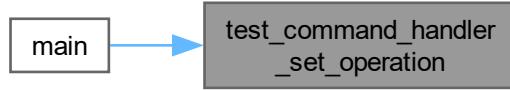
```
void test_command_handler_set_operation (
    void ) [extern]
```

Definition at line 39 of file [test_command_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.16 test_command_handler_invalid_operation()

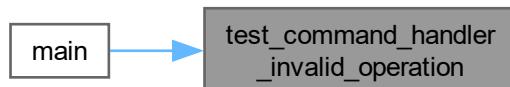
```
void test_command_handler_invalid_operation (
    void ) [extern]
```

Definition at line 54 of file [test_command_handlers.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.17 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
    void ) [extern]
```

Definition at line 6 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

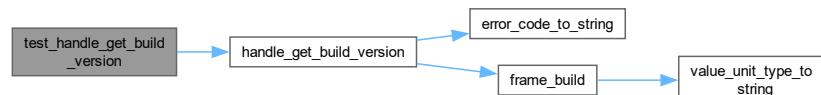


8.155.1.18 `test_handle_get_build_version()`

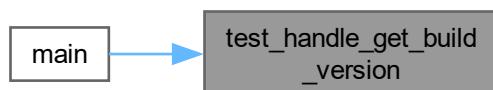
```
void test_handle_get_build_version (
    void ) [extern]
```

Definition at line 15 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

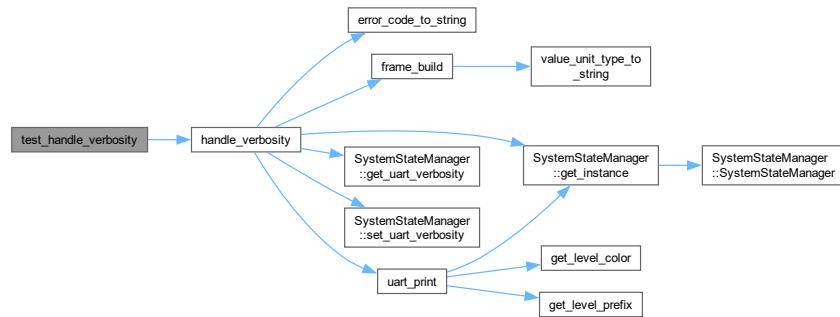


8.155.1.19 test_handle_verbosity()

```
void test_handle_verbosity (
    void ) [extern]
```

Definition at line 25 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

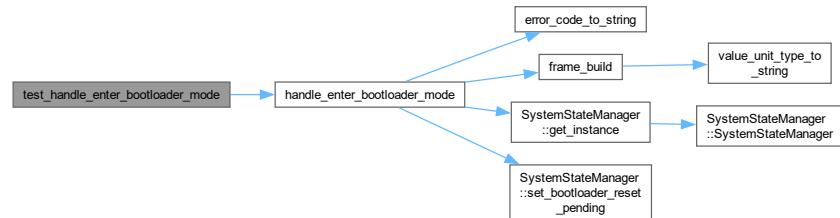


8.155.1.20 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
    void ) [extern]
```

Definition at line 40 of file [test_diagnostic_commands.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



8.155.1.21 test_error_code_conversion()

```
void test_error_code_conversion (
    void ) [extern]
```

Definition at line 101 of file [testMocks.cpp](#).

Here is the caller graph for this function:

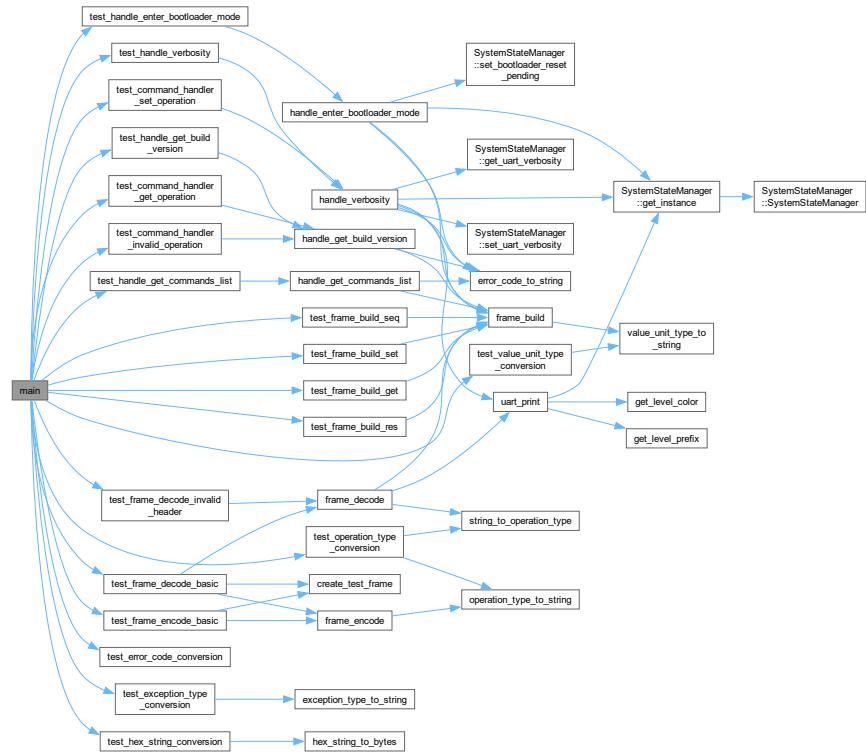


8.155.1.22 main()

```
int main (
    void )
```

Definition at line 38 of file [test_runner.cpp](#).

Here is the call graph for this function:



8.156 test_runner.cpp

[Go to the documentation of this file.](#)

```
00001 // test/test_runner.cpp
00002 #include "includes.h"
00003 #include "unity.h"
00004
00005 // External test function declarations
00006 // Pure software tests (no hardware dependencies)
00007 extern void test_frame_encode_basic(void);
00008 extern void test_frame_decode_basic(void);
00009 extern void test_frame_decode_invalid_header(void);
00010
00011 // FRAME BUILD
00012 extern void test_frame_build_get(void);
00013 extern void test_frame_build_set(void);
00014 extern void test_frame_build_res(void);
00015 extern void test_frame_build_seq(void);
00016 extern void test_frame_build_val(void);
00017 extern void test_frame_build_err(void);
00018
00019 extern void test_operation_type_conversion(void);
00020 extern void test_value_unit_type_conversion(void);
00021 extern void test_exception_type_conversion(void);
00022 extern void test_hex_string_conversion(void);
00023
00024 // Command handler tests
00025 extern void test_command_handler_get_operation(void);
00026 extern void test_command_handler_set_operation(void);
00027 extern void test_command_handler_invalid_operation(void);
00028
00029 //diagnostic
00030 extern void test_handle_get_commands_list(void);
00031 extern void test_handle_get_build_version(void);
00032 extern void test_handle_verbosity(void);
00033 extern void test_handle_enter_bootloader_mode(void);
00034
```

```
00035 // Error code tests
00036 extern void test_error_code_conversion(void);
00037
00038 int main(void) {
00039     stdio_init_all();
00040     uart_init(uart0, 115200);
00041     gpio_set_function(0, GPIO_FUNC_UART);
00042     gpio_set_function(1, GPIO_FUNC_UART);
00043
00044     UNITY_BEGIN();
00045     uart_puts(uart0, "begin unity tests\n");
00046
00047     // Frame codec tests (pure software)
00048     uart_puts(uart0, "begin frame codec tests\n");
00049     RUN_TEST(test_frame_encode_basic);
00050     RUN_TEST(test_frame_decode_basic);
00051     RUN_TEST(test_frame_decode_invalid_header);
00052     uart_puts(uart0, "end frame codec tests\n");
00053
00054     // Frame build tests (pure software)
00055     uart_puts(uart0, "begin frame build tests\n");
00056     RUN_TEST(test_frame_build_get);
00057     RUN_TEST(test_frame_build_set);
00058     RUN_TEST(test_frame_build_res);
00059     RUN_TEST(test_frame_build_seq);
00060     uart_puts(uart0, "end frame build tests\n");
00061
00062     // Converter tests (pure software)
00063     uart_puts(uart0, "begin converter tests\n");
00064     RUN_TEST(test_operation_type_conversion);
00065     RUN_TEST(test_value_unit_type_conversion);
00066     RUN_TEST(test_exception_type_conversion);
00067     RUN_TEST(test_hex_string_conversion);
00068     RUN_TEST(test_error_code_conversion);
00069     uart_puts(uart0, "end converter tests\n");
00070
00071     // Command handler tests (pure software)
00072     uart_puts(uart0, "begin command handler tests\n");
00073     RUN_TEST(test_command_handler_get_operation);
00074     RUN_TEST(test_command_handler_set_operation);
00075     RUN_TEST(test_command_handler_invalid_operation);
00076     uart_puts(uart0, "end command handler tests\n");
00077
00078     uart_puts(uart0, "begin diagnostic command handlers tests\n");
00079     RUN_TEST(test_handle_get_commands_list);
00080     RUN_TEST(test_handle_get_build_version);
00081     RUN_TEST(test_handle_verbosity);
00082     RUN_TEST(test_handle_enter_bootloader_mode);
00083     uart_puts(uart0, "end diagnostic commands handlers tests\n");
00084
00085     return UNITY_END();
00086 }
```

Index

_BH1750_DEFAULT_MTREG
 BH1750.h, [315](#)
_BH1750_DEVICE_ID
 BH1750.h, [314](#)
_BH1750_MTREG_MAX
 BH1750.h, [314](#)
_BH1750_MTREG_MIN
 BH1750.h, [314](#)
__attribute__
 Event Manager, [53](#), [55](#)
_filterRes
 INA3221, [155](#)
_i2c
 INA3221, [155](#)
_i2c_addr
 BH1750, [88](#)
 INA3221, [155](#)
_masken_reg
 INA3221, [155](#)
_read
 INA3221, [151](#)
_shuntRes
 INA3221, [155](#)
_write
 INA3221, [152](#)
~EventManager
 EventManager, [133](#)
~ISensor
 ISensor, [156](#)

ACCEL_X
 ISensor.h, [335](#)
ACCEL_Y
 ISensor.h, [335](#)
ACCEL_Z
 ISensor.h, [335](#)
ADDR_SDO_HIGH
 BME280, [95](#)
ADDR_SDO_LOW
 BME280, [95](#)
address
 HMC5883L, [146](#)
Alert Functions, [68](#)
 enable_alerts, [70](#)
 get_crit_alert, [71](#)
 get_power_valid_alert, [73](#)
 get_warn_alert, [71](#)
 set_alert_latch, [73](#)
 set_crit_alert_limit, [69](#)
 set_power_valid_limit, [70](#)

 set_warn_alert_limit, [69](#)
altitude
 TelemetryRecord, [191](#)
ANSI_BLUE
 utils.h, [368](#)
ANSI_GREEN
 utils.h, [368](#)
ANSI_RED
 utils.h, [368](#)
ANSI_RESET
 utils.h, [368](#)
ANSI_YELLOW
 utils.h, [368](#)
avg_mode
 INA3221::conf_reg_t, [108](#)

battery_voltage
 TelemetryRecord, [189](#)
bcd_to_bin
 DS3231, [124](#)
begin
 BH1750, [86](#)
 Configuration Functions, [57](#)
BH1750, [85](#)
 _i2c_addr, [88](#)
 begin, [86](#)
 BH1750, [86](#)
 configure, [86](#)
 CONTINUOUS_HIGH_RES_MODE, [86](#)
 CONTINUOUS_HIGH_RES_MODE_2, [86](#)
 CONTINUOUS_LOW_RES_MODE, [86](#)
 get_light_level, [87](#)
 Mode, [85](#)
 ONE_TIME_HIGH_RES_MODE, [86](#)
 ONE_TIME_HIGH_RES_MODE_2, [86](#)
 ONE_TIME_LOW_RES_MODE, [86](#)
 POWER_ON, [86](#)
 RESET, [86](#)
 UNCONFIGURED_POWER_DOWN, [86](#)
 write8, [87](#)
BH1750.h
 _BH1750_DEFAULT_MTREG, [315](#)
 _BH1750_DEVICE_ID, [314](#)
 _BH1750_MTREG_MAX, [314](#)
 _BH1750_MTREG_MIN, [314](#)
BH1750Wrapper, [88](#)
 BH1750Wrapper, [89](#)
 configure, [90](#)
 get_address, [90](#)
 get_i2c_addr, [89](#)

get_type, 90
 init, 89
 initialized_, 90
 is_initialized, 90
 read_data, 89
 sensor_, 90
 bin_to_bcd
 DS3231, 123
 BME280, 91
 ADDR_SDO_HIGH, 95
 ADDR_SDO_LOW, 95
 BME280, 93
 calib_params, 95
 configure_sensor, 94
 convert_humidity, 94
 convert_pressure, 94
 convert_temperature, 93
 device_addr, 95
 get_calibration_parameters, 94
 i2c_port, 95
 init, 93
 initialized_, 95
 NUM_CALIB_PARAMS, 100
 read_raw_all, 93
 REG_CONFIG, 95
 REG_CTRL_HUM, 96
 REG_CTRL_MEAS, 96
 REG_DIG_H1, 99
 REG_DIG_H2, 99
 REG_DIG_H3, 100
 REG_DIG_H4, 100
 REG_DIG_H5, 100
 REG_DIG_H6, 100
 REG_DIG_P1_LSB, 97
 REG_DIG_P1_MSB, 97
 REG_DIG_P2_LSB, 97
 REG_DIG_P2_MSB, 97
 REG_DIG_P3_LSB, 98
 REG_DIG_P3_MSB, 98
 REG_DIG_P4_LSB, 98
 REG_DIG_P4_MSB, 98
 REG_DIG_P5_LSB, 98
 REG_DIG_P5_MSB, 98
 REG_DIG_P6_LSB, 98
 REG_DIG_P6_MSB, 98
 REG_DIG_P7_LSB, 99
 REG_DIG_P7_MSB, 99
 REG_DIG_P8_LSB, 99
 REG_DIG_P8_MSB, 99
 REG_DIG_P9_LSB, 99
 REG_DIG_P9_MSB, 99
 REG_DIG_T1_LSB, 96
 REG_DIG_T1_MSB, 96
 REG_DIG_T2_LSB, 97
 REG_DIG_T2_MSB, 97
 REG_DIG_T3_LSB, 97
 REG_DIG_T3_MSB, 97
 REG_HUMIDITY_MSB, 96
 REG_PRESSURE_MSB, 96
 REG_RESET, 96
 REG_TEMPERATURE_MSB, 96
 reset, 93
 t_fine, 95
 BME280CalibParam, 100
 dig_h1, 103
 dig_h2, 103
 dig_h3, 103
 dig_h4, 103
 dig_h5, 103
 dig_h6, 103
 dig_p1, 101
 dig_p2, 102
 dig_p3, 102
 dig_p4, 102
 dig_p5, 102
 dig_p6, 102
 dig_p7, 102
 dig_p8, 102
 dig_p9, 102
 dig_t1, 101
 dig_t2, 101
 dig_t3, 101
 BME280Wrapper, 104
 BME280Wrapper, 105
 configure, 106
 get_address, 106
 get_type, 105
 init, 105
 initialized_, 106
 is_initialized, 105
 read_data, 105
 sensor_, 106
 BOOL
 protocol.h, 252
 BOOT
 Event Manager, 52
 buffer
 main.cpp, 375
 buffer_index
 main.cpp, 375
 BUFFER_SIZE
 pin_config.h, 292
 BUILD_NUMBER
 build_number.h, 193
 build_number.h, 193
 BUILD_NUMBER, 193
 build_version
 TelemetryRecord, 189
 bus_conv_time
 INA3221::conf_reg_t, 107
 calib_params
 BME280, 95
 century
 ds3231_data_t, 128
 ch1_en
 INA3221::conf_reg_t, 108

ch2_en
 INA3221::conf_reg_t, 108
ch3_en
 INA3221::conf_reg_t, 108
CHANGED
 Event Manager, 53
charge_current_solar
 TelemetryRecord, 189
charge_current_usb
 TelemetryRecord, 189
CHARGE_SOLAR
 power_commands.cpp, 223
CHARGE_TOTAL
 power_commands.cpp, 223
CHARGE_USB
 power_commands.cpp, 222
charging_solar_active_
 PowerManager, 170
charging_usb_active_
 PowerManager, 170
check_power_alerts
 PowerManager, 169
CLOCK
 Event Manager, 51
Clock Commands, 1
Clock Management Commands, 13
 handle_clock_sync_interval, 15
 handle_get_last_sync_time, 16
 handle_time, 13
 handle_timezone_offset, 14
 systemClock, 17
clock_commands.cpp
 CLOCK_GROUP, 206
 CLOCK_SYNC_INTERVAL, 206
 LAST_SYNC_TIME, 206
 TIME, 206
 TIMEZONE_OFFSET, 206
clock_enable
 DS3231, 115
CLOCK_GROUP
 clock_commands.cpp, 206
clock_mutex_
 DS3231, 125
CLOCK_SYNC_INTERVAL
 clock_commands.cpp, 206
ClockEvent
 Event Manager, 53
collect_gps_data
 gps_collector.cpp, 280
 gps_collector.h, 283
collect_telemetry
 Telemetry Manager, 76
command
 Frame, 142
Command System, 17
 command_handlers, 19
 CommandHandler, 17
 CommandMap, 17
 execute_command, 18
 command_handlers
 Command System, 19
 CommandAccessLevel
 protocol.h, 251
 CommandHandler
 Command System, 17
 frame.cpp, 246
 CommandMap
 Command System, 17
COMMS
 Event Manager, 51
CommsEvent
 Event Manager, 52
communication.cpp
 initialize_radio, 236
 interval, 238
 lastPrintTime, 238
 lastReceiveTime, 238
 lastSendTime, 238
 lora_tx_done_callback, 237
 msgCount, 238
 outgoing, 238
communication.h
 determine_unit, 244
 handle_uart_input, 242
 initialize_radio, 240
 lora_tx_done_callback, 241
 on_receive, 241
 send_frame_lora, 243
 send_frame_uart, 243
 send_message, 243
 split_and_send_message, 244
Configuration Functions, 56
 begin, 57
 get_die_id, 59
 get_manufacturer_id, 58
 INA3221, 57
 read_register, 60
 reset, 58
 set_averaging_mode, 64
 set_bus_conversion_time, 64
 set_bus_measurement_disable, 63
 set_bus_measurement_enable, 63
 set_mode_continuous, 61
 set_mode_power_down, 60
 set_mode_triggered, 61
 set_shunt_conversion_time, 65
 set_shunt_measurement_disable, 62
 set_shunt_measurement_enable, 62
configure
 BH1750, 86
 BH1750Wrapper, 90
 BME280Wrapper, 106
 HMC5883LWrapper, 148
 ISensor, 156
 MPU6050Wrapper, 161
 PowerManager, 168

configure_sensor
 BME280, 94
 CONTINUOUS_HIGH_RES_MODE
 BH1750, 86
 CONTINUOUS_HIGH_RES_MODE_2
 BH1750, 86
 CONTINUOUS_LOW_RES_MODE
 BH1750, 86
 conv_ready
 INA3221::masken_reg_t, 158
 convert_humidity
 BME280, 94
 convert_pressure
 BME280, 94
 convert_temperature
 BME280, 93
 core1_entry
 main.cpp, 372
 CORE1_START
 Event Manager, 52
 CORE1_STOP
 Event Manager, 52
 course
 TelemetryRecord, 191
 create_test_frame
 test_frame_common.h, 399
 crit_alert_ch1
 INA3221::masken_reg_t, 159
 crit_alert_ch2
 INA3221::masken_reg_t, 159
 crit_alert_ch3
 INA3221::masken_reg_t, 158
 crit_alert_latch_en
 INA3221::masken_reg_t, 159
 DATA_READY
 Event Manager, 53
 date
 ds3231_data_t, 128
 TelemetryRecord, 191
 DATETIME
 protocol.h, 252
 day
 ds3231_data_t, 127
 days_of_week
 DS3231.h, 203
 DEBUG
 utils.h, 368
 DEBUG_UART_BAUD_RATE
 pin_config.h, 290
 DEBUG_UART_PORT
 pin_config.h, 290
 DEBUG_UART_RX_PIN
 pin_config.h, 291
 DEBUG_UART_TX_PIN
 pin_config.h, 291
 DEFAULT_FLUSH_THRESHOLD
 telemetry_manager.cpp, 351
 DEFAULT_SAMPLE_INTERVAL_MS
 telemetry_manager.cpp, 351
 DELIMITER
 protocol.h, 258
 determine_unit
 communication.h, 244
 device_addr
 BME280, 95
 Diagnostic Commands, 19
 handle_enter_bootloader_mode, 23
 handle_get_build_version, 21
 handle_get_commands_list, 20
 handle_verbose, 22
 dig_h1
 BME280CalibParam, 103
 dig_h2
 BME280CalibParam, 103
 dig_h3
 BME280CalibParam, 103
 dig_h4
 BME280CalibParam, 103
 dig_h5
 BME280CalibParam, 103
 dig_h6
 BME280CalibParam, 103
 dig_p1
 BME280CalibParam, 101
 dig_p2
 BME280CalibParam, 102
 dig_p3
 BME280CalibParam, 102
 dig_p4
 BME280CalibParam, 102
 dig_p5
 BME280CalibParam, 102
 dig_p6
 BME280CalibParam, 102
 dig_p7
 BME280CalibParam, 102
 dig_p8
 BME280CalibParam, 102
 dig_p9
 BME280CalibParam, 102
 dig_t1
 BME280CalibParam, 101
 dig_t2
 BME280CalibParam, 101
 dig_t3
 BME280CalibParam, 101
 direction
 Frame, 142
 discharge_current
 TelemetryRecord, 189
 DRAW_TOTAL
 power_commands.cpp, 223
 DS3231, 108
 bcd_to_bin, 124
 bin_to_bcd, 123
 clock_enable, 115

clock_mutex_, 125
DS3231, 110
ds3231_addr, 125
get_clock_sync_interval, 117
get_last_sync_time, 118
get_local_time, 119
get_time, 111
get_timezone_offset, 116
get_unix_time, 114
i2c, 125
i2c_read_reg, 121
i2c_write_reg, 122
is_sync_needed, 120
last_sync_time_, 126
read_temperature, 112
set_clock_sync_interval, 117
set_time, 110
set_timezone_offset, 116
set_unix_time, 113
sync_clock_with_gps, 120
sync_interval_minutes_, 126
timezone_offset_minutes_, 126
update_last_sync_time, 118

DS3231.h
days_of_week, 203
DS3231_CONTROL_REG, 202
DS3231_CONTROL_STATUS_REG, 203
DS3231_DATE_REG, 202
DS3231_DAY_REG, 202
DS3231_DEVICE_ADRESS, 201
DS3231_HOURS_REG, 202
DS3231_MINUTES_REG, 201
DS3231_MONTH_REG, 202
DS3231_SECONDS_REG, 201
DS3231_TEMPERATURE_LSB_REG, 203
DS3231_TEMPERATURE_MSB_REG, 203
DS3231_YEAR_REG, 202
FRIDAY, 203
MONDAY, 203
SATURDAY, 203
SUNDAY, 203
THURSDAY, 203
TUESDAY, 203
WEDNESDAY, 203

ds3231_addr
DS3231, 125
DS3231_CONTROL_REG
DS3231.h, 202
DS3231_CONTROL_STATUS_REG
DS3231.h, 203
ds3231_data_t, 126
century, 128
date, 128
day, 127
hours, 127
minutes, 127
month, 128
seconds, 127

year, 128
DS3231_DATE_REG
DS3231.h, 202
DS3231_DAY_REG
DS3231.h, 202
DS3231_DEVICE_ADRESS
DS3231.h, 201
DS3231_HOURS_REG
DS3231.h, 202
DS3231_MINUTES_REG
DS3231.h, 201
DS3231_MONTH_REG
DS3231.h, 202
DS3231_SECONDS_REG
DS3231.h, 201
DS3231_TEMPERATURE_LSB_REG
DS3231.h, 203
DS3231_TEMPERATURE_MSB_REG
DS3231.h, 203
DS3231_YEAR_REG
DS3231.h, 202

emit
EventEmitter, 129

enable_alerts
Alert Functions, 70

ENVIRONMENT
ISensor.h, 335

ERR
protocol.h, 251

ERROR
Event Manager, 53
utils.h, 368

error_code_to_string
protocol.h, 253
utils_converters.cpp, 268

ErrorCode
protocol.h, 250

event
event_manager.h, 277
EventLog, 132

Event Commands, 25
handle_get_event_count, 26
handle_get_last_events, 25

Event Manager, 49
__attribute__, 53, 55
BOOT, 52
CHANGED, 53
CLOCK, 51
ClockEvent, 53
COMMS, 51
CommsEvent, 52
CORE1_START, 52
CORE1_STOP, 52
DATA_READY, 53
ERROR, 53
EventGroup, 51
eventLogId, 55
eventManager, 55

get_event, 54
 GPS, 51
 GPS_SYNC, 53
 GPS_SYNC_DATA_NOT_READY, 53
 GPSEvent, 52
 LOCK, 53
 log_event, 53
 LOST, 53
 LOW_BATTERY, 52
 MSG RECEIVED, 52
 MSG SENT, 52
 OVERCHARGE, 52
 PASS THROUGH END, 53
 PASS THROUGH START, 53
 POWER, 51
 POWER_FALLING, 52
 POWER_NORMAL, 52
 POWER_OFF, 53
 POWER_ON, 53
 PowerEvent, 52
 RADIO_ERROR, 52
 RADIO_INIT, 52
 SHUTDOWN, 52
 SOLAR_ACTIVE, 52
 SOLAR_INACTIVE, 52
 SYSTEM, 51
 systemClock, 55
 SystemEvent, 51
 UART_ERROR, 52
 USB_CONNECTED, 52
 USB_DISCONNECTED, 52
 WATCHDOG_RESET, 52
EVENT_BUFFER_SIZE
 event_manager.h, 276
EVENT_FLUSH_THRESHOLD
 event_manager.h, 276
EVENT_LOG_FILE
 event_manager.h, 276
event_manager.h
 event, 277
EVENT_BUFFER_SIZE, 276
EVENT_FLUSH_THRESHOLD, 276
EVENT_LOG_FILE, 276
 group, 277
 id, 276
 timestamp, 276
 to_string, 276
eventCount
 EventManager, 135
EventEmitter, 128
 emit, 129
EventGroup
 Event Manager, 51
EventLog, 130
 event, 132
 group, 131
 id, 131
 timestamp, 131
 to_string, 131
eventLogId
 Event Manager, 55
EventManager, 132
 ~EventManager, 133
eventCount, 135
EventManager, 133
eventMutex, 136
events, 135
eventsSinceFlush, 136
get_event_count, 134
init, 134
load_from_storage, 135
nextEventId, 136
save_to_storage, 134
writeIndex, 136
eventManager
 Event Manager, 55
EventManagerImpl, 137
 EventManagerImpl, 139
 load_from_storage, 139
 save_to_storage, 139
eventMutex
 EventManager, 136
eventRegister
 frame.cpp, 246
events
 EventManager, 135
eventsSinceFlush
 EventManager, 136
exception_type_to_string
 protocol.h, 252
 utils_converters.cpp, 266
ExceptionType
 protocol.h, 252
execute_command
 Command System, 18
FAIL_TO_SET
 protocol.h, 251
FALL_RATE_THRESHOLD
 PowerManager, 170
FALLING_TREND_REQUIRED
 PowerManager, 170
fd
 FileHandle, 140
FileHandle, 140
 fd, 140
 is_open, 140
fix_quality
 TelemetryRecord, 191
flush_sensor_data
 Telemetry Manager, 77
flush_telemetry
 Telemetry Manager, 76
flush_threshold
 telemetry_manager.cpp, 351
footer
 Frame, 143

Frame, 141
 command, 142
 direction, 142
 footer, 143
 group, 142
 header, 142
 operationType, 142
 unit, 143
 value, 143
Frame Handling, 44
 frame_build, 48
 frame_decode, 46
 frame_encode, 45
 frame_process, 47
frame.cpp
 CommandHandler, 246
 eventRegister, 246
FRAME_BEGIN
 protocol.h, 258
frame_build
 Frame Handling, 48
frame_decode
 Frame Handling, 46
frame_encode
 Frame Handling, 45
FRAME_END
 protocol.h, 258
frame_process
 Frame Handling, 47
FRIDAY
 DS3231.h, 203
fs_close_file
 storage.h, 344
fs_file_exists
 storage.h, 344
fs_init
 storage.cpp, 340
 storage.h, 343
fs_open_file
 storage.h, 343
fs_read_file
 storage.h, 344
fs_stop
 storage.cpp, 340
fs_write_file
 storage.h, 344
g_pending_bootloader_reset
 testMocks.cpp, 409
g_uart_verbosity
 testMocks.cpp, 409
GET
 protocol.h, 251
get_address
 BH1750Wrapper, 90
 BME280Wrapper, 106
 HMC5883LWrapper, 148
 ISensor, 157
get_available_sensors
 SensorWrapper, 177
get_calibration_parameters
 BME280, 94
get_clock_sync_interval
 DS3231, 117
get_crit_alert
 Alert Functions, 71
get_current
 INA3221, 154
get_current_charge_solar
 PowerManager, 167
get_current_charge_total
 PowerManager, 167
get_current_charge_usb
 PowerManager, 167
get_current_draw
 PowerManager, 168
get_current_ma
 Measurement Functions, 67
get_die_id
 Configuration Functions, 59
get_event
 Event Manager, 54
get_event_count
 EventManager, 134
get_gga_tokens
 NMEAData, 163
get_i2c_addr
 BH1750Wrapper, 89
get_instance
 SensorWrapper, 174
 SystemStateManager, 181
GET_LAST_SENSOR_RECORD_COMMAND
 telemetry_commands.cpp, 234
get_last_sensor_record_csv
 Telemetry Manager, 81
get_last_sync_time
 DS3231, 118
GET_LAST_TELEMETRY_RECORD_COMMAND
 telemetry_commands.cpp, 234
get_last_telemetry_record_csv
 Telemetry Manager, 80
get_level_color
 utils.cpp, 360
get_level_prefix
 utils.cpp, 361
get_light_level
 BH1750, 87
get_local_time
 DS3231, 119
get_manufacturer_id
 Configuration Functions, 58
get_power_valid_alert
 Alert Functions, 73
get_rmc_tokens
 NMEAData, 163
get_sensor
 SensorWrapper, 176

get_shunt_voltage
 Measurement Functions, 66

get_telemetry_flush_threshold
 Telemetry Manager, 80

get_telemetry_sample_interval
 Telemetry Manager, 79

get_time
 DS3231, 111

get_timezone_offset
 DS3231, 116

get_type
 BH1750Wrapper, 90

 BME280Wrapper, 105

 HMC5883LWrapper, 148

 ISensor, 156

 MPU6050Wrapper, 161

get_uart_verbosity
 SystemStateManager, 185

get_unix_time
 DS3231, 114

 NMEAData, 164

get_voltage
 Measurement Functions, 67

get_voltage_5v
 PowerManager, 168

get_voltage_battery
 PowerManager, 168

get_warn_alert
 Alert Functions, 71

GGA_DATA_COMMAND
 gps_commands.cpp, 218

gga_mutex_
 NMEAData, 165

gga_tokens_
 NMEAData, 164

GPS
 Event Manager, 51

GPS Commands, 27

- handle_enable_gps_uart_passthrough, 28
- handle_get_gga_data, 30
- handle_get_rmc_data, 29
- handle_gps_power_status, 27

gps_collection_paused
 SystemStateManager, 187

gps_collector.cpp
 collect_gps_data, 280

 MAX_RAW_DATA_LENGTH, 280

 nmea_data, 281

 splitString, 280

gps_collector.h
 collect_gps_data, 283

gps_commands.cpp
 GGA_DATA_COMMAND, 218

 GPS_GROUP, 218

 PASSTHROUGH_COMMAND, 218

 POWER_STATUS_COMMAND, 218

 RMC_DATA_COMMAND, 218

GPS_GROUP

gps_commands.cpp, 218

GPS_POWER_ENABLE_PIN
 pin_config.h, 292

GPS_SYNC
 Event Manager, 53

GPS_SYNC_DATA_NOT_READY
 Event Manager, 53

GPS_UART_BAUD_RATE
 pin_config.h, 291

GPS_UART_PORT
 pin_config.h, 291

GPS_UART_RX_PIN
 pin_config.h, 292

GPS_UART_TX_PIN
 pin_config.h, 291

GPSEvent
 Event Manager, 52

group
 event_manager.h, 277

 EventLog, 131

 Frame, 142

GYRO_X
 ISensor.h, 335

GYRO_Y
 ISensor.h, 335

GYRO_Z
 ISensor.h, 335

handle_clock_sync_interval
 Clock Management Commands, 15

handle_enable_gps_uart_passthrough
 GPS Commands, 28

handle_enter_bootloader_mode
 Diagnostic Commands, 23

handle_get_build_version
 Diagnostic Commands, 21

handle_get_commands_list
 Diagnostic Commands, 20

handle_get_current_charge_solar
 Power Commands, 34

handle_get_current_charge_total
 Power Commands, 35

handle_get_current_charge_usb
 Power Commands, 34

handle_get_current_draw
 Power Commands, 36

handle_get_event_count
 Event Commands, 26

handle_get_gga_data
 GPS Commands, 30

handle_get_last_events
 Event Commands, 25

handle_get_last_sensor_record
 Telemetry Buffer Commands, 43

handle_get_last_sync_time
 Clock Management Commands, 16

handle_get_last(telemetry_record
 Telemetry Buffer Commands, 43

handle_get_power_manager_ids

Power Commands, 31
handle_get_rmc_data
 GPS Commands, 29
handle_get_sensor_data
 Sensor Commands, 37
handle_get_sensor_list
 Sensor Commands, 39
handle_get_voltage_5v
 Power Commands, 33
handle_get_voltage_battery
 Power Commands, 32
handle_gps_power_status
 GPS Commands, 27
handle_list_files
 Storage Commands, 41
handle_mount
 Storage Commands, 41
handle_sensor_config
 Sensor Commands, 38
handle_time
 Clock Management Commands, 13
handle_timezone_offset
 Clock Management Commands, 14
handle_uart_input
 communication.h, 242
 receive.cpp, 261
handle_verbosity
 Diagnostic Commands, 22
hardwareMocks.cpp
 mock_spi_enabled, 403
 mock_spi_read_blocking, 403
 mock_spi_write_blocking, 403
 mock_uart_enabled, 403
 mock_uart_init, 402
 mock_uart_puts, 402
 spi_output_buffer, 403
 uart_output_buffer, 403
hardwareMocks.h
 mock_spi_enabled, 406
 mock_spi_read_blocking, 406
 mock_spi_write_blocking, 406
 mock_uart_enabled, 406
 mock_uart_init, 406
 mock_uart_puts, 406
 spi_output_buffer, 407
 uart_output_buffer, 406
has_valid_time
 NMEAData, 163
header
 Frame, 142
hex_string_to_bytes
 protocol.h, 255
 utils_converters.cpp, 270
HMC5883L, 143
 address, 146
 HMC5883L, 144
 i2c, 146
 init, 144
 read, 144
 read_register, 145
 write_register, 145
HMC5883LWrapper, 146
 configure, 148
 get_address, 148
 get_type, 148
 HMC5883LWrapper, 147
 init, 148
 initialized_, 149
 is_initialized, 148
 read_data, 148
 sensor_, 149
hours
 ds3231_data_t, 127
HUMIDITY
 ISensor.h, 335
humidity
 SensorDataRecord, 172
i2c
 DS3231, 125
 HMC5883L, 146
i2c_port
 BME280, 95
i2c_read_reg
 DS3231, 121
i2c_write_reg
 DS3231, 122
id
 event_manager.h, 276
 EventLog, 131
IMU
 ISensor.h, 335
INA3221, 149
 _filterRes, 155
 _i2c, 155
 _i2c_addr, 155
 _masken_reg, 155
 _read, 151
 _shuntRes, 155
 _write, 152
 Configuration Functions, 57
 get_current, 154
INA3221 Power Monitor, 56
INA3221.h
 INA3221_ADDR40_GND, 303
 INA3221_ADDR41_VCC, 303
 INA3221_ADDR42_SDA, 303
 INA3221_ADDR43_SCL, 303
 ina3221_addr_t, 303
 ina3221_avg_mode_t, 304
 INA3221_CH1, 303
 INA3221_CH2, 303
 INA3221_CH3, 303
 INA3221_CH_NUM, 305
 ina3221_ch_t, 303
 ina3221_conv_time_t, 304
 INA3221_REG_CH1_BUSV, 304

INA3221_REG_CH1_CRIT_ALERT_LIM, 304
 INA3221_REG_CH1_SHUNTV, 304
 INA3221_REG_CH1_WARNING_ALERT_LIM,
 304
 INA3221_REG_CH2_BUSV, 304
 INA3221_REG_CH2_CRIT_ALERT_LIM, 304
 INA3221_REG_CH2_SHUNTV, 304
 INA3221_REG_CH2_WARNING_ALERT_LIM,
 304
 INA3221_REG_CH3_BUSV, 304
 INA3221_REG_CH3_CRIT_ALERT_LIM, 304
 INA3221_REG_CH3_SHUNTV, 304
 INA3221_REG_CH3_WARNING_ALERT_LIM,
 304
 INA3221_REG_CONF, 304
 INA3221_REG_CONF_AVG_1, 305
 INA3221_REG_CONF_AVG_1024, 305
 INA3221_REG_CONF_AVG_128, 305
 INA3221_REG_CONF_AVG_16, 305
 INA3221_REG_CONF_AVG_256, 305
 INA3221_REG_CONF_AVG_4, 305
 INA3221_REG_CONF_AVG_512, 305
 INA3221_REG_CONF_AVG_64, 305
 INA3221_REG_CONF_CT_1100US, 304
 INA3221_REG_CONF_CT_140US, 304
 INA3221_REG_CONF_CT_204US, 304
 INA3221_REG_CONF_CT_2116US, 304
 INA3221_REG_CONF_CT_332US, 304
 INA3221_REG_CONF_CT_4156US, 304
 INA3221_REG_CONF_CT_588US, 304
 INA3221_REG_CONF_CT_8244US, 304
 INA3221_REG_DIE_ID, 304
 INA3221_REG_MANUF_ID, 304
 INA3221_REG_MASK_ENABLE, 304
 INA3221_REG_PWR_VALID_HI_LIM, 304
 INA3221_REG_PWR_VALID_LO_LIM, 304
 INA3221_REG_SHUNTV_SUM, 304
 INA3221_REG_SHUNTV_SUM_LIM, 304
 ina3221_reg_t, 303
 SHUNT_VOLTAGE_LSB_UV, 305
 INA3221::conf_reg_t, 106
 avg_mode, 108
 bus_conv_time, 107
 ch1_en, 108
 ch2_en, 108
 ch3_en, 108
 mode_bus_en, 107
 mode_continious_en, 107
 mode_shunt_en, 107
 reset, 108
 shunt_conv_time, 107
 INA3221::masken_reg_t, 157
 conv_ready, 158
 crit_alert_ch1, 159
 crit_alert_ch2, 159
 crit_alert_ch3, 158
 crit_alert_latch_en, 159
 pwr_valid_alert, 158
 reserved, 159
 shunt_sum_alert, 158
 shunt_sum_en_ch1, 159
 shunt_sum_en_ch2, 159
 shunt_sum_en_ch3, 159
 timing_ctrl_alert, 158
 warn_alert_ch1, 158
 warn_alert_ch2, 158
 warn_alert_ch3, 158
 warn_alert_latch_en, 159
 ina3221_
 PowerManager, 170
 INA3221_ADDR40_GND
 INA3221.h, 303
 INA3221_ADDR41_VCC
 INA3221.h, 303
 INA3221_ADDR42_SDA
 INA3221.h, 303
 INA3221_ADDR43_SCL
 INA3221.h, 303
 ina3221_addr_t
 INA3221.h, 303
 ina3221_avg_mode_t
 INA3221.h, 304
 INA3221_CH1
 INA3221.h, 303
 INA3221_CH2
 INA3221.h, 303
 INA3221_CH3
 INA3221.h, 303
 INA3221_CH_NUM
 INA3221.h, 305
 ina3221_ch_t
 INA3221.h, 303
 ina3221_conv_time_t
 INA3221.h, 304
 INA3221_REG_CH1_BUSV
 INA3221.h, 304
 INA3221_REG_CH1_CRIT_ALERT_LIM
 INA3221.h, 304
 INA3221_REG_CH1_SHUNTV
 INA3221.h, 304
 INA3221_REG_CH1_WARNING_ALERT_LIM
 INA3221.h, 304
 INA3221_REG_CH2_BUSV
 INA3221.h, 304
 INA3221_REG_CH2_CRIT_ALERT_LIM
 INA3221.h, 304
 INA3221_REG_CH2_SHUNTV
 INA3221.h, 304
 INA3221_REG_CH2_WARNING_ALERT_LIM
 INA3221.h, 304
 INA3221_REG_CH3_BUSV
 INA3221.h, 304
 INA3221_REG_CH3_CRIT_ALERT_LIM
 INA3221.h, 304
 INA3221_REG_CH3_SHUNTV
 INA3221.h, 304

INA3221_REG_CH3_WARNING_ALERT_LIM
 INA3221.h, 304

INA3221_REG_CONF
 INA3221.h, 304

INA3221_REG_CONF_AVG_1
 INA3221.h, 305

INA3221_REG_CONF_AVG_1024
 INA3221.h, 305

INA3221_REG_CONF_AVG_128
 INA3221.h, 305

INA3221_REG_CONF_AVG_16
 INA3221.h, 305

INA3221_REG_CONF_AVG_256
 INA3221.h, 305

INA3221_REG_CONF_AVG_4
 INA3221.h, 305

INA3221_REG_CONF_AVG_512
 INA3221.h, 305

INA3221_REG_CONF_AVG_64
 INA3221.h, 305

INA3221_REG_CONF_CT_1100US
 INA3221.h, 304

INA3221_REG_CONF_CT_140US
 INA3221.h, 304

INA3221_REG_CONF_CT_204US
 INA3221.h, 304

INA3221_REG_CONF_CT_2116US
 INA3221.h, 304

INA3221_REG_CONF_CT_332US
 INA3221.h, 304

INA3221_REG_CONF_CT_4156US
 INA3221.h, 304

INA3221_REG_CONF_CT_588US
 INA3221.h, 304

INA3221_REG_CONF_CT_8244US
 INA3221.h, 304

INA3221_REG_DIE_ID
 INA3221.h, 304

INA3221_REG_MANUF_ID
 INA3221.h, 304

INA3221_REG_MASK_ENABLE
 INA3221.h, 304

INA3221_REG_PWR_VALID_HI_LIM
 INA3221.h, 304

INA3221_REG_PWR_VALID_LO_LIM
 INA3221.h, 304

INA3221_REG_SHUNTV_SUM
 INA3221.h, 304

INA3221_REG_SHUNTV_SUM_LIM
 INA3221.h, 304

ina3221_reg_t
 INA3221.h, 303

includes.h, 194

INFO
 utils.h, 368

init
 BH1750Wrapper, 89

 BME280, 93

BME280Wrapper, 105

EventManager, 134

HMC5883L, 144

HMC5883LWrapper, 148

ISensor, 156

MPU6050Wrapper, 161

init_modules
 main.cpp, 373

init_pico_hw
 main.cpp, 372

initialize
 PowerManager, 167

initialize_radio
 communication.cpp, 236

 communication.h, 240

initialized_
 BH1750Wrapper, 90

 BME280, 95

 BME280Wrapper, 106

 HMC5883LWrapper, 149

 MPU6050Wrapper, 162

 PowerManager, 170

instance
 SystemStateManager, 186

instance_mutex
 SystemStateManager, 186

Interface
 protocol.h, 252

INTERNAL_FAIL_TO_READ
 protocol.h, 251

interval
 communication.cpp, 238

INVALID_FORMAT
 protocol.h, 251

INVALID_OPERATION
 protocol.h, 251, 252

INVALID_PARAM
 protocol.h, 252

INVALID_VALUE
 protocol.h, 251

is_bootloader_reset_pending
 SystemStateManager, 182

is_charging_solar
 PowerManager, 168

is_charging_usb
 PowerManager, 168

is_gps_collection_paused
 SystemStateManager, 183

is_initialized
 BH1750Wrapper, 90

 BME280Wrapper, 105

 HMC5883LWrapper, 148

 ISensor, 156

 MPU6050Wrapper, 161

is_open
 FileHandle, 140

is_sd_card_mounted
 SystemStateManager, 184

is_sync_needed
 DS3231, 120

is_telemetry_collection_time
 Telemetry Manager, 78

is_telemetry_flush_time
 Telemetry Manager, 79

ISensor, 155

- ~ISensor, 156
- configure, 156
- get_address, 157
- get_type, 156
- init, 156
- is_initialized, 156
- read_data, 156

ISensor.h

- ACCEL_X, 335
- ACCEL_Y, 335
- ACCEL_Z, 335
- ENVIRONMENT, 335
- GYRO_X, 335
- GYRO_Y, 335
- GYRO_Z, 335
- HUMIDITY, 335
- IMU, 335
- LIGHT, 335
- LIGHT_LEVEL, 335
- MAG_FIELD_X, 335
- MAG_FIELD_Y, 335
- MAG_FIELD_Z, 335
- MAGNETOMETER, 335
- PRESSURE, 335
- SensorDataTypelIdentifier, 335
- SensorType, 335
- TEMPERATURE, 335

last_frame_sent
 test_command_handlers.cpp, 387

LAST_SYNC_TIME
 clock_commands.cpp, 206

last_sync_time_
 DS3231, 126

lastPrintTime
 communication.cpp, 238

lastReceiveTime
 communication.cpp, 238

lastSendTime
 communication.cpp, 238

lat_dir
 TelemetryRecord, 190

latitude
 TelemetryRecord, 190

lib/clock/DS3231.cpp, 195

lib/clock/DS3231.h, 200, 204

lib/comms/commands/clock_commands.cpp, 205, 207

lib/comms/commands/commands.cpp, 209, 210

lib/comms/commands/commands.h, 211, 213

lib/comms/commands/diagnostic_commands.cpp, 213, 214

lib/comms/commands/event_commands.cpp, 216

lib/comms/commands/gps_commands.cpp, 217, 219

lib/comms/commands/power_commands.cpp, 221, 223

lib/comms/commands/sensor_commands.cpp, 225, 227

lib/comms/commands/storage_commands.cpp, 230, 232

lib/comms/commands/telemetry_commands.cpp, 233, 235

lib/comms/communication.cpp, 236, 239

lib/comms/communication.h, 239, 245

lib/comms/frame.cpp, 245, 246

lib/comms/protocol.h, 248, 258

lib/comms/receive.cpp, 259, 262

lib/comms/send.cpp, 263, 265

lib/comms/utils_converters.cpp, 265, 271

lib/eventman/event_manager.cpp, 272, 273

lib/eventman/event_manager.h, 274, 277

lib/location/gps_collector.cpp, 279, 281

lib/location/gps_collector.h, 282, 284

lib/location/NMEA/NMEA_data.cpp, 284, 285

lib/location/NMEA/NMEA_data.h, 286, 287

lib/pin_config.cpp, 288, 289

lib/pin_config.h, 289, 296

lib/powerman/INA3221/INA3221.cpp, 296, 297

lib/powerman/INA3221/INA3221.h, 301, 306

lib/powerman/PowerManager.cpp, 308

lib/powerman/PowerManager.h, 310, 311

lib/sensors/BH1750/BH1750.cpp, 312

lib/sensors/BH1750/BH1750.h, 313, 315

lib/sensors/BH1750/BH1750_WRAPPER.cpp, 315, 316

lib/sensors/BH1750/BH1750_WRAPPER.h, 317, 318

lib/sensors/BME280/BME280.cpp, 318, 319

lib/sensors/BME280/BME280.h, 322, 323

lib/sensors/BME280/BME280_WRAPPER.cpp, 325

lib/sensors/BME280/BME280_WRAPPER.h, 326

lib/sensors/HMC5883L/HMC5883L.cpp, 327

lib/sensors/HMC5883L/HMC5883L.h, 328, 329

lib/sensors/HMC5883L/HMC5883L_WRAPPER.cpp, 329, 330

lib/sensors/HMC5883L/HMC5883L_WRAPPER.h, 331, 332

lib/sensors/ISensor.cpp, 332, 333

lib/sensors/ISensor.h, 334, 336

lib/sensors/MPU6050/MPU6050.cpp, 336

lib/sensors/MPU6050/MPU6050.h, 337

lib/sensors/MPU6050/MPU6050_WRAPPER.cpp, 337

lib/sensors/MPU6050/MPU6050_WRAPPER.h, 337, 338

lib/storage/storage.cpp, 339, 341

lib/storage/storage.h, 342, 345

lib/system_state_manager.cpp, 345, 346

lib/system_state_manager.h, 347, 348

lib/telemetry/telemetry_manager.cpp, 349, 352

lib/telemetry/telemetry_manager.h, 356, 358

lib/utils.cpp, 359, 366

lib/utils.h, 366, 370

LIGHT
 ISensor.h, 335

light

SensorDataRecord, 172
LIGHT_LEVEL
 ISensor.h, 335
LIST_FILES_COMMAND
 storage_commands.cpp, 231
load_from_storage
 EventManager, 135
 EventManagerImpl, 139
LOCK
 Event Manager, 53
log_event
 Event Manager, 53
LOG_FILENAME
 main.cpp, 372
lon_dir
 TelemetryRecord, 190
longitude
 TelemetryRecord, 190
LORA
 protocol.h, 252
lora_address_local
 pin_config.cpp, 288
 pin_config.h, 295
lora_address_remote
 pin_config.cpp, 289
 pin_config.h, 295
lora_cs_pin
 pin_config.cpp, 288
 pin_config.h, 295
LORA_DEFAULT_DIO0_PIN
 pin_config.h, 294
LORA_DEFAULT_RESET_PIN
 pin_config.h, 294
LORA_DEFAULT_SPI
 pin_config.h, 293
LORA_DEFAULT_SPI_FREQUENCY
 pin_config.h, 294
LORA_DEFAULT_SS_PIN
 pin_config.h, 294
lora_irq_pin
 pin_config.cpp, 288
 pin_config.h, 295
lora_reset_pin
 pin_config.cpp, 288
 pin_config.h, 295
lora_send_called
 test_command_handlers.cpp, 386
lora_tx_done_callback
 communication.cpp, 237
 communication.h, 241
LOST
 Event Manager, 53
LOW_BATTERY
 Event Manager, 52
MAG_FIELD_X
 ISensor.h, 335
MAG_FIELD_Y
 ISensor.h, 335
MAG_FIELD_Z
 ISensor.h, 335
MAGNETOMETER
 ISensor.h, 335
main
 main.cpp, 374
 test_runner.cpp, 424
main.cpp, 371
 buffer, 375
 buffer_index, 375
 core1_entry, 372
 init_modules, 373
 init_pico_hw, 372
 LOG_FILENAME, 372
 main, 374
 powerManager, 374
 systemClock, 374
MAIN_I2C_PORT
 pin_config.h, 291
MAIN_I2C_SCL_PIN
 pin_config.h, 291
MAIN_I2C_SDA_PIN
 pin_config.h, 291
MAX_PACKET_SIZE
 receive.cpp, 260
MAX_RAW_DATA_LENGTH
 gps_collector.cpp, 280
Measurement Functions, 66
 get_current_ma, 67
 get_shunt_voltage, 66
 get_voltage, 67
MILIAMP
 protocol.h, 252
minutes
 ds3231_data_t, 127
mock_spi_enabled
 hardwareMocks.cpp, 403
 hardwareMocks.h, 406
mock_spi_read_blocking
 hardwareMocks.cpp, 403
 hardwareMocks.h, 406
mock_spi_write_blocking
 hardwareMocks.cpp, 403
 hardwareMocks.h, 406
mock_uart_enabled
 hardwareMocks.cpp, 403
 hardwareMocks.h, 406
 testMocks.cpp, 410
mock_uart_init
 hardwareMocks.cpp, 402
 hardwareMocks.h, 406
mock_uart_puts
 hardwareMocks.cpp, 402
 hardwareMocks.h, 406
Mode
 BH1750, 85
mode_bus_en
 INA3221::conf_reg_t, 107

mode_continious_en
 INA3221::conf_reg_t, 107
 mode_shunt_en
 INA3221::conf_reg_t, 107
 MONDAY
 DS3231.h, 203
 month
 ds3231_data_t, 128
 MOUNT_COMMAND
 storage_commands.cpp, 231
 MPU6050Wrapper, 160
 configure, 161
 get_type, 161
 init, 161
 initialized_, 162
 is_initialized, 161
 MPU6050Wrapper, 161
 read_data, 161
 sensor_, 162
 MSG_RECEIVED
 Event Manager, 52
 MSG_SENT
 Event Manager, 52
 msgCount
 communication.cpp, 238

 nextEventId
 EventManager, 136
 nmea_data
 gps_collector.cpp, 281
 NMEA_data.cpp, 285
 NMEA_data.h, 287
 telemetry_manager.cpp, 351
 testMocks.cpp, 409
 NMEA_data.cpp
 nmea_data, 285
 NMEA_data.h
 nmea_data, 287
 NMEAData, 162
 get_gga_tokens, 163
 get_rmc_tokens, 163
 get_unix_time, 164
 gga_mutex_, 165
 gga_tokens_, 164
 has_valid_time, 163
 NMEAData, 163
 rmc_mutex_, 164
 rmc_tokens_, 164
 update_gga_tokens, 163
 update_rmc_tokens, 163
 NONE
 protocol.h, 251, 252
 NOT_ALLOWED
 protocol.h, 251, 252
 NUM_CALIB_PARAMS
 BME280, 100

 on_receive
 communication.h, 241

 receive.cpp, 260
 ONE_TIME_HIGH_RES_MODE
 BH1750, 86
 ONE_TIME_HIGH_RES_MODE_2
 BH1750, 86
 ONE_TIME_LOW_RES_MODE
 BH1750, 86
 operation_type_to_string
 protocol.h, 254
 utils_converters.cpp, 267
 OperationType
 protocol.h, 251
 operationType
 Frame, 142
 operator=
 SystemStateManager, 186
 outgoing
 communication.cpp, 238
 OVERCHARGE
 Event Manager, 52

 PA_OUTPUT_PA_BOOST_PIN
 pin_config.h, 294
 PA_OUTPUT_RFO_PIN
 pin_config.h, 294
 PARAM_INVALID
 protocol.h, 251
 PARAM_REQUIRED
 protocol.h, 251
 PARAM_UNNECESSARY
 protocol.h, 252
 PARAM_UNNECESSARY
 protocol.h, 251
 PASS_THROUGH_END
 Event Manager, 53
 PASS_THROUGH_START
 Event Manager, 53
 PASSTHROUGH_COMMAND
 gps_commands.cpp, 218
 pending_bootloader_reset
 SystemStateManager, 186
 pin_config.cpp
 lora_address_local, 288
 lora_address_remote, 289
 lora_cs_pin, 288
 lora_irq_pin, 288
 lora_reset_pin, 288
 pin_config.h
 BUFFER_SIZE, 292
 DEBUG_UART_BAUD_RATE, 290
 DEBUG_UART_PORT, 290
 DEBUG_UART_RX_PIN, 291
 DEBUG_UART_TX_PIN, 291
 GPS_POWER_ENABLE_PIN, 292
 GPS_UART_BAUD_RATE, 291
 GPS_UART_PORT, 291
 GPS_UART_RX_PIN, 292
 GPS_UART_TX_PIN, 291
 lora_address_local, 295

lora_address_remote, 295
lora_cs_pin, 295
LORA_DEFAULT_DIO0_PIN, 294
LORA_DEFAULT_RESET_PIN, 294
LORA_DEFAULT_SPI, 293
LORA_DEFAULT_SPI_FREQUENCY, 294
LORA_DEFAULT_SS_PIN, 294
lora_irq_pin, 295
lora_reset_pin, 295
MAIN_I2C_PORT, 291
MAIN_I2C_SCL_PIN, 291
MAIN_I2C_SDA_PIN, 291
PA_OUTPUT_PA_BOOST_PIN, 294
PA_OUTPUT_RFO_PIN, 294
READ_BIT, 293
SD_CARD_DETECT_PIN, 293
SD_CS_PIN, 292
SD_MISO_PIN, 292
SD_MOSI_PIN, 292
SD_SCK_PIN, 292
SD_SPI_PORT, 292
SPI_PORT, 293
SX1278_CS, 293
SX1278_MISO, 293
SX1278_MOSI, 293
SX1278_SCK, 293

POWER
Event Manager, 51

Power Commands, 31

- handle_get_current_charge_solar, 34
- handle_get_current_charge_total, 35
- handle_get_current_charge_usb, 34
- handle_get_current_draw, 36
- handle_get_power_manager_ids, 31
- handle_get_voltage_5v, 33
- handle_get_voltage_battery, 32

power_commands.cpp

- CHARGE_SOLAR, 223
- CHARGE_TOTAL, 223
- CHARGE_USB, 222
- DRAW_TOTAL, 223
- POWER_GROUP, 222
- POWER_MANAGER_IDS, 222
- VOLTAGE_BATTERY, 222
- VOLTAGE_MAIN, 222

POWER_FALLING
Event Manager, 52

POWER_GROUP

- power_commands.cpp, 222

POWER_MANAGER_IDS

- power_commands.cpp, 222

POWER_NORMAL
Event Manager, 52

POWER_OFF
Event Manager, 53

POWER_ON

- BH1750, 86
- Event Manager, 53

POWER_STATUS_COMMAND
gps_commands.cpp, 218

PowerEvent
Event Manager, 52

powerman_mutex_
PowerManager, 170

PowerManager, 165

- charging_solar_active_, 170
- charging_usb_active_, 170
- check_power_alerts, 169
- configure, 168
- FALL_RATE_THRESHOLD, 170
- FALLING_TREND_REQUIRED, 170
- get_current_charge_solar, 167
- get_current_charge_total, 167
- get_current_charge_usb, 167
- get_current_draw, 168
- get_voltage_5v, 168
- get_voltage_battery, 168
- ina3221_, 170
- initialize, 167
- initialized_, 170
- is_charging_solar, 168
- is_charging_usb, 168
- powerman_mutex_, 170
- PowerManager, 166
- read_device_ids, 167
- SOLAR_CURRENT_THRESHOLD, 169
- USB_CURRENT_THRESHOLD, 169
- VOLTAGE_LOW_THRESHOLD, 169
- VOLTAGE_OVERCHARGE_THRESHOLD, 170

powerManager

- main.cpp, 374
- telemetry_manager.cpp, 351
- testMocks.cpp, 409

PRESSURE
ISensor.h, 335

pressure
SensorDataRecord, 172

protocol.h

- BOOL, 252
- CommandAccessLevel, 251
- DATETIME, 252
- DELIMITER, 258
- ERR, 251
- error_code_to_string, 253
- ErrorCode, 250
- exception_type_to_string, 252
- ExceptionType, 252
- FAIL_TO_SET, 251
- FRAME_BEGIN, 258
- FRAME_END, 258
- GET, 251
- hex_string_to_bytes, 255
- Interface, 252
- INTERNAL_FAIL_TO_READ, 251
- INVALID_FORMAT, 251
- INVALID_OPERATION, 251, 252

INVALID_PARAM, 252
 INVALID_VALUE, 251
 LORA, 252
 MILIAMP, 252
 NONE, 251, 252
 NOT_ALLOWED, 251, 252
 operation_type_to_string, 254
 OperationType, 251
 PARAM_INVALID, 251
 PARAM_REQUIRED, 251
 PARAM_UNNECESSARY, 252
 PARAM_UNNECESSARY, 251
 READ_ONLY, 251
 READ_WRITE, 251
 RES, 251
 SECOND, 252
 SEQ, 251
 SET, 251
 string_to_operation_type, 255
 TEXT, 252
 UART, 252
 UNDEFINED, 252
 UNKNOWN_ERROR, 251
 VAL, 251
 value_unit_type_to_string, 256
 ValueUnit, 251
 VOLT, 252
 WRITE_ONLY, 251
 pwr_valid_alert
 INA3221::masken_reg_t, 158

RADIO_ERROR
 Event Manager, 52

RADIO_INIT
 Event Manager, 52

read
 HMC5883L, 144

READ_BIT
 pin_config.h, 293

read_data
 BH1750Wrapper, 89
 BME280Wrapper, 105
 HMC5883LWrapper, 148
 ISensor, 156
 MPU6050Wrapper, 161

read_device_ids
 PowerManager, 167

READ_ONLY
 protocol.h, 251

read_raw_all
 BME280, 93

read_register
 Configuration Functions, 60
 HMC5883L, 145

read_temperature
 DS3231, 112

READ_WRITE
 protocol.h, 251

receive.cpp

handle_uart_input, 261
 MAX_PACKET_SIZE, 260
 on_receive, 260

REG_CONFIG
 BME280, 95

REG_CTRL_HUM
 BME280, 96

REG_CTRL_MEAS
 BME280, 96

REG_DIG_H1
 BME280, 99

REG_DIG_H2
 BME280, 99

REG_DIG_H3
 BME280, 100

REG_DIG_H4
 BME280, 100

REG_DIG_H5
 BME280, 100

REG_DIG_H6
 BME280, 100

REG_DIG_P1_LSB
 BME280, 97

REG_DIG_P1_MSB
 BME280, 97

REG_DIG_P2_LSB
 BME280, 97

REG_DIG_P2_MSB
 BME280, 97

REG_DIG_P3_LSB
 BME280, 98

REG_DIG_P3_MSB
 BME280, 98

REG_DIG_P4_LSB
 BME280, 98

REG_DIG_P4_MSB
 BME280, 98

REG_DIG_P5_LSB
 BME280, 98

REG_DIG_P5_MSB
 BME280, 98

REG_DIG_P6_LSB
 BME280, 98

REG_DIG_P6_MSB
 BME280, 98

REG_DIG_P7_LSB
 BME280, 99

REG_DIG_P7_MSB
 BME280, 99

REG_DIG_P8_LSB
 BME280, 99

REG_DIG_P8_MSB
 BME280, 99

REG_DIG_P9_LSB
 BME280, 99

REG_DIG_P9_MSB
 BME280, 99

REG_DIG_T1_LSB

BME280, 96
REG_DIG_T1_MSB
 BME280, 96
REG_DIG_T2_LSB
 BME280, 97
REG_DIG_T2_MSB
 BME280, 97
REG_DIG_T3_LSB
 BME280, 97
REG_DIG_T3_MSB
 BME280, 97
REG_HUMIDITY_MSB
 BME280, 96
REG_PRESSURE_MSB
 BME280, 96
REG_RESET
 BME280, 96
REG_TEMPERATURE_MSB
 BME280, 96
RES
 protocol.h, 251
reserved
 INA3221::masken_reg_t, 159
RESET
 BH1750, 86
reset
 BME280, 93
 Configuration Functions, 58
 INA3221::conf_reg_t, 108
RMIC_DATA_COMMAND
 gps_commands.cpp, 218
rmc_mutex_
 NMEAData, 164
rmc_tokens_
 NMEAData, 164

sample_interval_ms
 telemetry_manager.cpp, 351
satellites
 TelemetryRecord, 191
SATURDAY
 DS3231.h, 203
save_to_storage
 EventManager, 134
 EventManagerImpl, 139
scan_connected_sensors
 SensorWrapper, 177
SD_CARD_DETECT_PIN
 pin_config.h, 293
sd_card_mounted
 storage.h, 344
 SystemStateManager, 187
SD_CS_PIN
 pin_config.h, 292
SD_MISO_PIN
 pin_config.h, 292
SD_MOSI_PIN
 pin_config.h, 292
SD_SCK_PIN
 pin_config.h, 292
SD_SPI_PORT
 pin_config.h, 292
SECOND
 protocol.h, 252
seconds
 ds3231_data_t, 127
send.cpp
 send_frame_lora, 264
 send_frame_uart, 264
 send_message, 263
send_frame_lora
 communication.h, 243
 send.cpp, 264
 test_comand_handlers.cpp, 383
send_frame_uart
 communication.h, 243
 send.cpp, 264
 test_comand_handlers.cpp, 383
send_message
 communication.h, 243
 send.cpp, 263
Sensor Commands, 37
 handle_get_sensor_data, 37
 handle_get_sensor_list, 39
 handle_sensor_config, 38
sensor_
 BH1750Wrapper, 90
 BME280Wrapper, 106
 HMC5883LWrapper, 149
 MPU6050Wrapper, 162
sensor_commands.cpp
 SENSOR_CONFIGURE, 226
 SENSOR_GROUP, 226
 SENSOR_READ, 226
SENSOR_CONFIGURE
 sensor_commands.cpp, 226
sensor_configure
 SensorWrapper, 175
sensor_data_buffer
 Telemetry Manager, 83
sensor_data_buffer_count
 Telemetry Manager, 83
sensor_data_buffer_write_index
 Telemetry Manager, 83
SENSOR_DATA_CSV_PATH
 telemetry_manager.cpp, 350
SENSOR_GROUP
 sensor_commands.cpp, 226
sensor_init
 SensorWrapper, 175
SENSOR_READ
 sensor_commands.cpp, 226
sensor_read_data
 SensorWrapper, 176
SensorDataRecord, 171
 humidity, 172
 light, 172

pressure, 172
 temperature, 172
 timestamp, 172
 to_csv, 171
SensorDataTypelIdentifier
 ISensor.h, 335
sensors
 SensorWrapper, 178
SensorType
 ISensor.h, 335
SensorWrapper, 173
 get_available_sensors, 177
 get_instance, 174
 get_sensor, 176
 scan_connected_sensors, 177
 sensor_configure, 175
 sensor_init, 175
 sensor_read_data, 176
 sensors, 178
 SensorWrapper, 174
SEQ
 protocol.h, 251
SET
 protocol.h, 251
set_alert_latch
 Alert Functions, 73
set_averaging_mode
 Configuration Functions, 64
set_bootloader_reset_pending
 SystemStateManager, 182
set_bus_conversion_time
 Configuration Functions, 64
set_bus_measurement_disable
 Configuration Functions, 63
set_bus_measurement_enable
 Configuration Functions, 63
set_clock_sync_interval
 DS3231, 117
set_crit_alert_limit
 Alert Functions, 69
set_gps_collection_paused
 SystemStateManager, 183
set_mode_continuous
 Configuration Functions, 61
set_mode_power_down
 Configuration Functions, 60
set_mode_triggered
 Configuration Functions, 61
set_power_valid_limit
 Alert Functions, 70
set_sd_card_mounted
 SystemStateManager, 184
set_shunt_conversion_time
 Configuration Functions, 65
set_shunt_measurement_disable
 Configuration Functions, 62
set_shunt_measurement_enable
 Configuration Functions, 62
set_telemetry_flush_threshold
 Telemetry Manager, 80
set_telemetry_sample_interval
 Telemetry Manager, 80
set_time
 DS3231, 110
set_timezone_offset
 DS3231, 116
set_uart_verbosity
 SystemStateManager, 185
set_unix_time
 DS3231, 113
set_warn_alert_limit
 Alert Functions, 69
setUp
 test_comand_handlers.cpp, 384
 test_frame_send.cpp, 400
shunt_conv_time
 INA3221::conf_reg_t, 107
shunt_sum_alert
 INA3221::masken_reg_t, 158
shunt_sum_en_ch1
 INA3221::masken_reg_t, 159
shunt_sum_en_ch2
 INA3221::masken_reg_t, 159
shunt_sum_en_ch3
 INA3221::masken_reg_t, 159
SHUNT_VOLTAGE_LSB_UV
 INA3221.h, 305
SHUTDOWN
 Event Manager, 52
SILENT
 utils.h, 368
SOLAR_ACTIVE
 Event Manager, 52
SOLAR_CURRENT_THRESHOLD
 PowerManager, 169
SOLAR_INACTIVE
 Event Manager, 52
speed
 TelemetryRecord, 190
spi_output_buffer
 hardwareMocks.cpp, 403
 hardwareMocks.h, 407
SPI_PORT
 pin_config.h, 293
split_and_send_message
 communication.h, 244
splitString
 gps_collector.cpp, 280
Storage Commands, 40
 handle_list_files, 41
 handle_mount, 41
storage.cpp
 fs_init, 340
 fs_stop, 340
storage.h
 fs_close_file, 344

fs_file_exists, 344
fs_init, 343
fs_open_file, 343
fs_read_file, 344
fs_write_file, 344
sd_card_mounted, 344
storage_commands.cpp
 LIST_FILES_COMMAND, 231
 MOUNT_COMMAND, 231
 STORAGE_GROUP, 231
STORAGE_GROUP
 storage_commands.cpp, 231
string_to_operation_type
 protocol.h, 255
 utils_converters.cpp, 268
SUNDAY
 DS3231.h, 203
SX1278_CS
 pin_config.h, 293
SX1278_MISO
 pin_config.h, 293
SX1278_MOSI
 pin_config.h, 293
SX1278_SCK
 pin_config.h, 293
sync_clock_with_gps
 DS3231, 120
sync_interval_minutes_
 DS3231, 126
SYSTEM
 Event Manager, 51
system_voltage
 TelemetryRecord, 189
systemClock
 Clock Management Commands, 17
 Event Manager, 55
 main.cpp, 374
 telemetry_manager.cpp, 351
 testMocks.cpp, 409
SystemEvent
 Event Manager, 51
SystemStateManager, 178
 get_instance, 181
 get_uart_verbosity, 185
 gps_collection_paused, 187
 instance, 186
 instance_mutex, 186
 is_bootloader_reset_pending, 182
 is_gps_collection_paused, 183
 is_sd_card_mounted, 184
 operator=, 186
 pending_bootloader_reset, 186
 sd_card_mounted, 187
 set_bootloader_reset_pending, 182
 set_gps_collection_paused, 183
 set_sd_card_mounted, 184
 set_uart_verbosity, 185
SystemStateManager, 179, 180
 uart_verbosity, 187
t_fine
 BME280, 95
tearDown
 test_comand_handlers.cpp, 384
 test_frame_send.cpp, 400
Telemetry Buffer Commands, 42
 handle_get_last_sensor_record, 43
 handle_get_last_telemetry_record, 43
Telemetry Manager, 74
 collect_telemetry, 76
 flush_sensor_data, 77
 flush_telemetry, 76
 get_last_sensor_record_csv, 81
 get_last_telemetry_record_csv, 80
 get_telemetry_flush_threshold, 80
 get_telemetry_sample_interval, 79
 is_telemetry_collection_time, 78
 is_telemetry_flush_time, 79
 sensor_data_buffer, 83
 sensor_data_buffer_count, 83
 sensor_data_buffer_write_index, 83
 set_telemetry_flush_threshold, 80
 set_telemetry_sample_interval, 80
 telemetry_buffer, 82
 telemetry_buffer_count, 82
 TELEMETRY_BUFFER_SIZE, 82, 83
 telemetry_buffer_write_index, 82
 telemetry_init, 75
 telemetry_mutex, 83
telemetry_buffer
 Telemetry Manager, 82
telemetry_buffer_count
 Telemetry Manager, 82
 telemetry_commands.cpp, 234
TELEMETRY_BUFFER_SIZE
 Telemetry Manager, 82, 83
telemetry_buffer_write_index
 Telemetry Manager, 82
 telemetry_commands.cpp, 235
telemetry_commands.cpp
 GET_LAST_SENSOR_RECORD_COMMAND,
 234
 GET_LAST_TELEMETRY_RECORD_COMMAND,
 234
 telemetry_buffer_count, 234
 telemetry_buffer_write_index, 235
 TELEMETRY_GROUP, 234
 telemetry_mutex, 234
TELEMETRY_CSV_PATH
 telemetry_manager.cpp, 350
TELEMETRY_GROUP
 telemetry_commands.cpp, 234
telemetry_init
 Telemetry Manager, 75
telemetry_manager.cpp
 DEFAULT_FLUSH_THRESHOLD, 351
 DEFAULT_SAMPLE_INTERVAL_MS, 351

flush_threshold, 351
 nmea_data, 351
 powerManager, 351
 sample_interval_ms, 351
 SENSOR_DATA_CSV_PATH, 350
 systemClock, 351
 TELEMETRY_CSV_PATH, 350
telemetry_mutex
 Telemetry Manager, 83
 telemetry_commands.cpp, 234
TelemetryRecord, 187
 altitude, 191
 battery_voltage, 189
 build_version, 189
 charge_current_solar, 189
 charge_current_usb, 189
 course, 191
 date, 191
 discharge_current, 189
 fix_quality, 191
 lat_dir, 190
 latitude, 190
 lon_dir, 190
 longitude, 190
 satellites, 191
 speed, 190
 system_voltage, 189
 time, 190
 timestamp, 189
 to_csv, 188
TEMPERATURE
 ISensor.h, 335
temperature
 SensorDataRecord, 172
test/comms/commands/test_clock_commands.cpp, 377
test/comms/commands/test_diagnostic_commands.cpp,
 377, 380
test/comms/commands/test_event_commands.cpp, 381
test/comms/commands/test_gps_commands.cpp, 381
test/comms/commands/test_power_commands.cpp,
 381
test/comms/commands/test_sensor_commands.cpp,
 382
test/comms/commands/test_storage_commands.cpp,
 382
test/comms/commands/test_telemetry_commands.cpp,
 382
test/comms/test_comand_handlers.cpp, 382, 387
test/comms/test_converters.cpp, 388, 390
test/comms/test_frame_build.cpp, 391, 395
test/comms/test_frame_coding.cpp, 396, 398
test/comms/test_frame_common.h, 398, 400
test/comms/test_frame_send.cpp, 400, 401
test/mock/hardware_mock.cpp, 402, 404
test/mock/hardware_mock.h, 404, 407
test/testMocks.cpp, 407, 410
test/test_runner.cpp, 411, 425
test_comand_handlers.cpp
last_frame_sent, 387
lora_send_called, 386
send_frame_lora, 383
send_frame_uart, 383
setUp, 384
tearDown, 384
test_command_handler_get_operation, 384
test_command_handler_invalid_operation, 386
test_command_handler_set_operation, 385
uart_send_called, 386
test_command_handler_get_operation
 test_comand_handlers.cpp, 384
 test_mocks.cpp, 408
 test_runner.cpp, 419
test_command_handler_invalid_operation
 test_comand_handlers.cpp, 386
 test_mocks.cpp, 409
 test_runner.cpp, 421
test_command_handler_set_operation
 test_comand_handlers.cpp, 385
 test_mocks.cpp, 409
 test_runner.cpp, 420
test_converters.cpp
 test_exception_type_conversion, 389
 test_hex_string_conversion, 390
 test_operation_type_conversion, 388
 test_value_unit_type_conversion, 388
test_diagnostic_commands.cpp
 test_handle_enter_bootloader_mode, 380
 test_handle_get_build_version, 378
 test_handle_get_commands_list, 378
 test_handle_verbosity, 379
test_error_code_conversion
 test_mocks.cpp, 408
 test_runner.cpp, 424
test_exception_type_conversion
 test_converters.cpp, 389
 test_runner.cpp, 418
test_frame_build.cpp
 test_frame_build_err, 392
 test_frame_build_get, 392
 test_frame_build_res, 393
 test_frame_build_seq, 394
 test_frame_build_set, 393
 test_frame_build_val, 391
test_frame_build_err
 test_frame_build.cpp, 392
 test_runner.cpp, 416
test_frame_build_get
 test_frame_build.cpp, 392
 test_runner.cpp, 414
test_frame_build_res
 test_frame_build.cpp, 393
 test_runner.cpp, 415
test_frame_build_seq
 test_frame_build.cpp, 394
 test_runner.cpp, 415
test_frame_build_set

test_frame_build.cpp, 393
test_runner.cpp, 414
test_frame_build_val
 test_frame_build.cpp, 391
 test_runner.cpp, 416
test_frame_coding.cpp
 test_frame_decode_basic, 396
 test_frame_decode_invalid_header, 397
 test_frame_encode_basic, 396
test_frame_common.h
 create_test_frame, 399
test_frame_decode_basic
 test_frame_coding.cpp, 396
 test_runner.cpp, 412
test_frame_decode_invalid_header
 test_frame_coding.cpp, 397
 test_runner.cpp, 413
test_frame_encode_basic
 test_frame_coding.cpp, 396
 test_runner.cpp, 412
test_frame_send.cpp
 setUp, 400
 tearDown, 400
 test_send_frame_uart, 401
test_handle_enter_bootloader_mode
 test_diagnostic_commands.cpp, 380
 test_runner.cpp, 423
test_handle_get_build_version
 test_diagnostic_commands.cpp, 378
 test_runner.cpp, 422
test_handle_get_commands_list
 test_diagnostic_commands.cpp, 378
 test_runner.cpp, 421
test_handle_verbosity
 test_diagnostic_commands.cpp, 379
 test_runner.cpp, 422
test_hex_string_conversion
 test_converters.cpp, 390
 test_runner.cpp, 419
testMocks.cpp
 g_pending_bootloader_reset, 409
 g_uart_verbosity, 409
 mock_uart_enabled, 410
 nmea_data, 409
 powerManager, 409
 systemClock, 409
 test_command_handler_get_operation, 408
 test_command_handler_invalid_operation, 409
 test_command_handler_set_operation, 409
 test_error_code_conversion, 408
 uart_output_buffer, 410
 uart_print, 408
test_operation_type_conversion
 test_converters.cpp, 388
 test_runner.cpp, 417
test_runner.cpp
 main, 424
 test_command_handler_get_operation, 419
 test_command_handler_invalid_operation, 421
 test_command_handler_set_operation, 420
 test_error_code_conversion, 424
 test_exception_type_conversion, 418
 test_frame_build_err, 416
 test_frame_build_get, 414
 test_frame_build_res, 415
 test_frame_build_seq, 415
 test_frame_build_set, 414
 test_frame_build_val, 416
 test_frame_decode_basic, 412
 test_frame_decode_invalid_header, 413
 test_frame_encode_basic, 412
 test_handle_enter_bootloader_mode, 423
 test_handle_get_build_version, 422
 test_handle_get_commands_list, 421
 test_handle_verbosity, 422
 test_hex_string_conversion, 419
 test_operation_type_conversion, 417
 test_value_unit_type_conversion, 417
testSendFrameUart
 test_frame_send.cpp, 401
testValueUnitTypeConversion
 test_converters.cpp, 388
 test_runner.cpp, 417
TEXT
 protocol.h, 252
THURSDAY
 DS3231.h, 203
TIME
 clock_commands.cpp, 206
time
 TelemetryRecord, 190
timestamp
 event_manager.h, 276
 EventLog, 131
 SensorDataRecord, 172
 TelemetryRecord, 189
TIMEZONE_OFFSET
 clock_commands.cpp, 206
timezone_offset_minutes_
 DS3231, 126
timing_ctrl_alert
 INA3221::masken_reg_t, 158
to_csv
 SensorDataRecord, 171
 TelemetryRecord, 188
to_string
 event_manager.h, 276
 EventLog, 131
TUESDAY
 DS3231.h, 203
UART
 protocol.h, 252
UART_ERROR
 Event Manager, 52
uart_mutex
 utils.cpp, 365

uart_output_buffer
hardwareMocks.cpp, 403
hardwareMocks.h, 406
testMocks.cpp, 410
uart_print
testMocks.cpp, 408
utils.cpp, 363
utils.h, 369
uart_send_called
testCommandHandlers.cpp, 386
uart_verbosity
SystemStateManager, 187
UNCONFIGURED_POWER_DOWN
BH1750, 86
UNDEFINED
protocol.h, 252
unit
Frame, 143
UNKNOWN_ERROR
protocol.h, 251
update_gga_tokens
NMEAData, 163
update_last_sync_time
DS3231, 118
update_rmc_tokens
NMEAData, 163
USB_CONNECTED
Event Manager, 52
USB_CURRENT_THRESHOLD
PowerManager, 169
USB_DISCONNECTED
Event Manager, 52
utils.cpp
get_level_color, 360
get_level_prefix, 361
uart_mutex, 365
uart_print, 363
utils.h
ANSI_BLUE, 368
ANSI_GREEN, 368
ANSI_RED, 368
ANSI_RESET, 368
ANSI_YELLOW, 368
DEBUG, 368
ERROR, 368
INFO, 368
SILENT, 368
uart_print, 369
VerbosityLevel, 368
WARNING, 368
utils_converters.cpp
error_code_to_string, 268
exception_type_to_string, 266
hex_string_to_bytes, 270
operation_type_to_string, 267
string_to_operation_type, 268
value_unit_type_to_string, 266

protocol.h, 251
value
Frame, 143
value_unit_type_to_string
protocol.h, 256
utils_converters.cpp, 266
ValueUnit
protocol.h, 251
VerbosityLevel
utils.h, 368
VOLT
protocol.h, 252
VOLTAGE_BATTERY
power_commands.cpp, 222
VOLTAGE_LOW_THRESHOLD
PowerManager, 169
VOLTAGE_MAIN
power_commands.cpp, 222
VOLTAGE_OVERCHARGE_THRESHOLD
PowerManager, 170

warn_alert_ch1
INA3221::masken_reg_t, 158
warn_alert_ch2
INA3221::masken_reg_t, 158
warn_alert_ch3
INA3221::masken_reg_t, 158
warn_alert_latch_en
INA3221::masken_reg_t, 159
WARNING
utils.h, 368
WATCHDOG_RESET
Event Manager, 52
WEDNESDAY
DS3231.h, 203
write8
BH1750, 87
WRITE_ONLY
protocol.h, 251
write_register
HMC5883L, 145
writeIndex
EventManager, 136

year
ds3231_data_t, 128

VAL