

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	11
6.1 Clock Management Commands	11
6.1.1 Detailed Description	11
6.1.2 Function Documentation	11
6.1.2.1 handle_time()	11
6.1.2.2 handle_timezone_offset()	12
6.1.2.3 handle_clock_sync_interval()	12
6.1.2.4 handle_get_last_sync_time()	13
6.2 Command System	13
6.2.1 Detailed Description	14
6.2.2 Typedef Documentation	14
6.2.2.1 CommandHandler	14
6.2.2.2 CommandMap	14
6.2.3 Function Documentation	14
6.2.3.1 execute_command()	14
6.2.4 Variable Documentation	15
6.2.4.1 command_handlers	15
6.3 Diagnostic Commands	16
6.3.1 Detailed Description	16
6.3.2 Function Documentation	16
6.3.2.1 handle_get_commands_list()	16
6.3.2.2 handle_get_build_version()	16
6.3.2.3 handle_verbosity()	17
6.3.2.4 handle_enter_bootloader_mode()	17
6.4 Event Commands	18
6.4.1 Detailed Description	18
6.4.2 Function Documentation	18
6.4.2.1 handle_get_last_events()	18
6.4.2.2 handle_get_event_count()	19
6.5 GPS Commands	20

6.5.1 Detailed Description	20
6.5.2 Function Documentation	20
6.5.2.1 handle_gps_power_status()	20
6.5.2.2 handle_enable_gps_uart_passthrough()	21
6.5.2.3 handle_get_rmc_data()	21
6.5.2.4 handle_get_gga_data()	22
6.6 Power Commands	22
6.6.1 Detailed Description	23
6.6.2 Function Documentation	23
6.6.2.1 handle_get_power_manager_ids()	23
6.6.2.2 handle_get_voltage_battery()	23
6.6.2.3 handle_get_voltage_5v()	24
6.6.2.4 handle_get_current_charge_usb()	24
6.6.2.5 handle_get_current_charge_solar()	25
6.6.2.6 handle_get_current_charge_total()	25
6.6.2.7 handle_get_current_draw()	26
6.7 Sensor Commands	27
6.7.1 Detailed Description	27
6.7.2 Function Documentation	27
6.7.2.1 handle_get_sensor_data()	27
6.7.2.2 handle_sensor_config()	28
6.7.2.3 handle_get_sensor_list()	28
6.8 Storage Commands	29
6.8.1 Detailed Description	29
6.8.2 Function Documentation	29
6.8.2.1 handle_list_files()	29
6.8.2.2 handle_mount()	29
6.9 Telemetry Buffer Commands	30
6.9.1 Detailed Description	30
6.9.2 Function Documentation	30
6.9.2.1 handle_get_last_telemetry_record()	30
6.9.2.2 handle_get_last_sensor_record()	31
6.10 Frame Handling	31
6.10.1 Detailed Description	32
6.10.2 Function Documentation	32
6.10.2.1 frame_encode()	32
6.10.2.2 frame_decode()	32
6.10.2.3 frame_process()	33
6.10.2.4 frame_build()	33
6.11 Protocol	34
6.11.1 Detailed Description	34
6.11.2 Enumeration Type Documentation	34

6.11.2.1 ErrorCode	34
6.11.2.2 OperationType	35
6.11.2.3 CommandAccessLevel	35
6.11.2.4 ValueUnit	35
6.11.2.5 ExceptionType	36
6.11.2.6 Interface	36
6.12 Receiving Data	36
6.12.1 Detailed Description	37
6.12.2 Function Documentation	37
6.12.2.1 on_receive()	37
6.12.2.2 handle_uart_input()	37
6.13 Utility Converters	38
6.13.1 Detailed Description	38
6.13.2 Function Documentation	38
6.13.2.1 exception_type_to_string()	38
6.13.2.2 value_unit_type_to_string()	38
6.13.2.3 operation_type_to_string()	39
6.13.2.4 string_to_operation_type()	39
6.13.2.5 error_code_to_string()	39
6.13.2.6 hex_string_to_bytes()	39
6.14 RTC clock	40
6.14.1 Detailed Description	41
6.14.2 Function Documentation	41
6.14.2.1 DS3231()	41
6.14.2.2 get_instance()	41
6.14.2.3 set_time()	41
6.14.2.4 get_time()	42
6.14.2.5 read_temperature()	43
6.14.2.6 set_unix_time()	43
6.14.2.7 get_unix_time()	44
6.14.2.8 clock_enable()	44
6.14.2.9 get_timezone_offset()	45
6.14.2.10 set_timezone_offset()	45
6.14.2.11 get_clock_sync_interval()	45
6.14.2.12 set_clock_sync_interval()	45
6.14.2.13 get_last_sync_time()	46
6.14.2.14 update_last_sync_time()	46
6.14.2.15 get_local_time()	47
6.14.2.16 is_sync_needed()	47
6.14.2.17 sync_clock_with_gps()	47
6.14.2.18 i2c_read_reg()	48
6.14.2.19 i2c_write_reg()	49

6.14.2.20 bin_to_bcd()	49
6.14.2.21 bcd_to_bin()	50
6.15 Event Management	51
6.15.1 Detailed Description	52
6.15.2 Macro Definition Documentation	52
6.15.2.1 EVENT_BUFFER_SIZE	52
6.15.2.2 EVENT_FLUSH_THRESHOLD	52
6.15.2.3 EVENT_LOG_FILE	52
6.15.3 Enumeration Type Documentation	52
6.15.3.1 EventGroup	52
6.15.3.2 SystemEvent	53
6.15.3.3 PowerEvent	53
6.15.3.4 CommsEvent	53
6.15.3.5 GPSEvent	54
6.15.3.6 ClockEvent	54
6.15.4 Function Documentation	55
6.15.4.1 __attribute__()	55
6.15.4.2 init()	55
6.15.4.3 log_event()	55
6.15.4.4 get_event()	55
6.15.4.5 save_to_storage()	56
6.15.5 Variable Documentation	56
6.15.5.1 __attribute__	56
6.16 Location	56
6.16.1 Detailed Description	56
6.16.2 Macro Definition Documentation	57
6.16.2.1 MAX_RAW_DATA_LENGTH	57
6.16.3 Function Documentation	57
6.16.3.1 splitString()	57
6.16.3.2 collect_gps_data()	57
6.17 INA3221 Power Monitor	57
6.17.1 Detailed Description	58
6.17.2 Configuration Functions	58
6.17.2.1 Detailed Description	58
6.17.2.2 Function Documentation	58
6.17.3 Measurement Functions	62
6.17.3.1 Detailed Description	62
6.17.3.2 Function Documentation	62
6.18 Power Management	63
6.18.1 Detailed Description	64
6.18.2 Function Documentation	64
6.18.2.1 PowerManager()	64

6.18.2.2 <code>get_instance()</code>	65
6.18.2.3 <code>initialize()</code>	65
6.18.2.4 <code>read_device_ids()</code>	65
6.18.2.5 <code>get_voltage_battery()</code>	65
6.18.2.6 <code>get_voltage_5v()</code>	66
6.18.2.7 <code>get_current_charge_usb()</code>	66
6.18.2.8 <code>get_current_draw()</code>	66
6.18.2.9 <code>get_current_charge_solar()</code>	66
6.18.2.10 <code>get_current_charge_total()</code>	67
6.18.2.11 <code>configure()</code>	67
6.18.2.12 <code>is_charging_solar()</code>	68
6.18.2.13 <code>is_charging_usb()</code>	68
6.19 BH1750 Light Sensor	68
6.19.1 Detailed Description	69
6.19.2 Function Documentation	69
6.19.2.1 <code>BH1750()</code>	69
6.19.2.2 <code>begin()</code>	69
6.19.2.3 <code>configure()</code>	69
6.19.2.4 <code>get_light_level()</code>	70
6.19.2.5 <code>write8()</code>	70
6.19.3 Constants	70
6.19.3.1 Detailed Description	71
6.19.3.2 Macro Definition Documentation	71
6.19.4 Types	71
6.19.4.1 Detailed Description	72
6.19.4.2 Enumeration Type Documentation	72
6.20 Sensors	72
6.20.1 Detailed Description	73
6.20.2 Enumeration Type Documentation	73
6.20.2.1 <code>SensorType</code>	73
6.20.2.2 <code>SensorDataTypeIdentifier</code>	73
6.20.3 Function Documentation	74
6.20.3.1 <code>sensor_init()</code>	74
6.20.3.2 <code>sensor_configure()</code>	74
6.20.3.3 <code>sensor_read_data()</code>	74
6.20.3.4 <code>get_sensor()</code>	75
6.20.3.5 <code>scan_connected_sensors()</code>	75
6.20.3.6 <code>get_available_sensors()</code>	76
6.21 Storage	76
6.21.1 Detailed Description	76
6.21.2 Function Documentation	76
6.21.2.1 <code>fs_init()</code>	76

6.21.2.2 fs_stop()	77
6.22 System State Manager	77
6.22.1 Detailed Description	77
6.23 Telemetry Manager	77
6.23.1 Detailed Description	78
6.23.2 Macro Definition Documentation	78
6.23.2.1 TELEMETRY_CSV_PATH	78
6.23.2.2 SENSOR_DATA_CSV_PATH	78
6.23.2.3 DEFAULT_SAMPLE_INTERVAL_MS	79
6.23.2.4 DEFAULT_FLUSH_THRESHOLD	79
6.23.3 Function Documentation	79
6.23.3.1 to_csv() [1/2]	79
6.23.3.2 to_csv() [2/2]	79
6.23.3.3 collect_power_telemetry()	79
6.23.3.4 emit_power_events()	80
6.23.3.5 collect_gps_telemetry()	80
6.23.3.6 collect_sensor_telemetry()	80
6.23.3.7 TelemetryManager()	81
6.23.3.8 init()	81
6.23.3.9 collect_telemetry()	81
6.23.3.10 flush_telemetry()	82
6.23.3.11 is_telemetry_collection_time()	82
6.23.3.12 is_telemetry_flush_time()	82
6.23.3.13 get_last_telemetry_record_csv()	83
6.23.3.14 get_last_sensor_record_csv()	83
7 Class Documentation	85
7.1 BH1750 Class Reference	85
7.1.1 Detailed Description	86
7.1.2 Member Data Documentation	86
7.1.2.1 _i2c_addr	86
7.1.2.2 i2c_port_	86
7.2 BH1750Wrapper Class Reference	86
7.2.1 Detailed Description	87
7.2.2 Constructor & Destructor Documentation	87
7.2.2.1 BH1750Wrapper() [1/2]	87
7.2.2.2 BH1750Wrapper() [2/2]	87
7.2.3 Member Function Documentation	88
7.2.3.1 get_i2c_addr()	88
7.2.3.2 init()	88
7.2.3.3 read_data()	88
7.2.3.4 is_initialized()	88

7.2.3.5 <code>get_type()</code>	89
7.2.3.6 <code>configure()</code>	89
7.2.3.7 <code>get_address()</code>	89
7.2.4 Member Data Documentation	90
7.2.4.1 <code>sensor_</code>	90
7.2.4.2 <code>initialized_</code>	90
7.3 BME280 Class Reference	90
7.3.1 Detailed Description	92
7.3.2 Member Enumeration Documentation	92
7.3.2.1 anonymous enum	92
7.3.2.2 Oversampling	92
7.3.2.3 anonymous enum	92
7.3.2.4 anonymous enum	93
7.3.2.5 anonymous enum	94
7.3.3 Constructor & Destructor Documentation	94
7.3.3.1 <code>BME280()</code>	94
7.3.4 Member Function Documentation	94
7.3.4.1 <code>init()</code>	94
7.3.4.2 <code>reset()</code>	95
7.3.4.3 <code>read_raw_all()</code>	95
7.3.4.4 <code>convert_temperature()</code>	95
7.3.4.5 <code>convert_pressure()</code>	95
7.3.4.6 <code>convert_humidity()</code>	96
7.3.4.7 <code>write_register()</code>	96
7.3.4.8 <code>read_register()</code> [1/2]	96
7.3.4.9 <code>read_register()</code> [2/2]	97
7.3.4.10 <code>configure_sensor()</code>	97
7.3.4.11 <code>get_calibration_parameters()</code>	98
7.3.5 Member Data Documentation	98
7.3.5.1 <code>i2c_port</code>	98
7.3.5.2 <code>device_addr</code>	98
7.3.5.3 <code>calib_params</code>	98
7.3.5.4 <code>initialized_</code>	98
7.3.5.5 <code>t_fine</code>	99
7.4 BME280CalibParam Struct Reference	99
7.4.1 Detailed Description	100
7.4.2 Member Data Documentation	100
7.4.2.1 <code>dig_t1</code>	100
7.4.2.2 <code>dig_t2</code>	100
7.4.2.3 <code>dig_t3</code>	100
7.4.2.4 <code>dig_p1</code>	100
7.4.2.5 <code>dig_p2</code>	100

7.4.2.6 dig_p3	101
7.4.2.7 dig_p4	101
7.4.2.8 dig_p5	101
7.4.2.9 dig_p6	101
7.4.2.10 dig_p7	101
7.4.2.11 dig_p8	101
7.4.2.12 dig_p9	102
7.4.2.13 dig_h1	102
7.4.2.14 dig_h2	102
7.4.2.15 dig_h3	102
7.4.2.16 dig_h4	102
7.4.2.17 dig_h5	102
7.4.2.18 dig_h6	103
7.5 BME280Wrapper Class Reference	103
7.5.1 Detailed Description	104
7.5.2 Constructor & Destructor Documentation	104
7.5.2.1 BME280Wrapper()	104
7.5.3 Member Function Documentation	104
7.5.3.1 init()	104
7.5.3.2 read_data()	104
7.5.3.3 is_initialized()	105
7.5.3.4 get_type()	105
7.5.3.5 configure()	105
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_continuous_en	107
7.6.2.4 shunt_conv_time	107
7.6.2.5 bus_conv_time	107
7.6.2.6 avg_mode	107
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() [1/2]	110
7.7.2.2 DS3231() [2/2]	110
7.7.3 Member Function Documentation	110
7.7.3.1 operator=()	110
7.7.4 Member Data Documentation	111
7.7.4.1 i2c	111
7.7.4.2 ds3231_addr	111
7.7.4.3 clock_mutex_	111
7.7.4.4 timezone_offset_minutes_	111
7.7.4.5 sync_interval_minutes_	111
7.7.4.6 last_sync_time_	111
7.8 ds3231_data_t Struct Reference	112
7.8.1 Detailed Description	112
7.8.2 Member Data Documentation	112
7.8.2.1 seconds	112
7.8.2.2 minutes	112
7.8.2.3 hours	113
7.8.2.4 day	113
7.8.2.5 date	113
7.8.2.6 month	113
7.8.2.7 year	113
7.8.2.8 century	113
7.9 EventEmitter Class Reference	114
7.9.1 Detailed Description	114
7.9.2 Member Function Documentation	114
7.9.2.1 emit()	114
7.10 EventLog Class Reference	115
7.10.1 Detailed Description	115
7.10.2 Member Data Documentation	115
7.10.2.1 id	115
7.10.2.2 timestamp	115
7.10.2.3 group	115
7.10.2.4 event	116
7.11 EventManager Class Reference	116
7.11.1 Detailed Description	117
7.11.2 Constructor & Destructor Documentation	117
7.11.2.1 EventManager() [1/2]	117
7.11.2.2 EventManager() [2/2]	117
7.11.3 Member Function Documentation	117
7.11.3.1 operator=()	117
7.11.3.2 get_instance()	117

7.11.3.3	get_event_count()	118
7.11.3.4	load_from_storage()	118
7.11.4	Member Data Documentation	118
7.11.4.1	events	118
7.11.4.2	eventCount	118
7.11.4.3	writeIndex	118
7.11.4.4	eventMutex	118
7.11.4.5	nextEventId	119
7.11.4.6	eventsSinceFlush	119
7.12	Frame Struct Reference	119
7.12.1	Detailed Description	119
7.12.2	Member Data Documentation	120
7.12.2.1	header	120
7.12.2.2	direction	120
7.12.2.3	operationType	120
7.12.2.4	group	120
7.12.2.5	command	120
7.12.2.6	value	120
7.12.2.7	unit	120
7.12.2.8	footer	121
7.13	INA3221 Class Reference	121
7.13.1	Detailed Description	122
7.13.2	Member Function Documentation	122
7.13.2.1	_read()	122
7.13.2.2	_write()	123
7.13.2.3	get_current()	123
7.13.3	Member Data Documentation	123
7.13.3.1	_i2c_addr	123
7.13.3.2	_i2c	123
7.13.3.3	_shuntRes	123
7.13.3.4	_filterRes	124
7.13.3.5	_masken_reg	124
7.14	ISensor Class Reference	124
7.14.1	Detailed Description	125
7.14.2	Constructor & Destructor Documentation	125
7.14.2.1	~ISensor()	125
7.14.3	Member Function Documentation	125
7.14.3.1	init()	125
7.14.3.2	read_data()	125
7.14.3.3	is_initialized()	126
7.14.3.4	get_type()	126
7.14.3.5	configure()	126

7.14.3.6 <code>get_address()</code>	126
7.15 <code>INA3221::maskn_reg_t</code> Struct Reference	127
7.15.1 Detailed Description	127
7.15.2 Member Data Documentation	127
7.15.2.1 <code>conv_ready</code>	127
7.15.2.2 <code>timing_ctrl_alert</code>	127
7.15.2.3 <code>pwr_valid_alert</code>	127
7.15.2.4 <code>warn_alert_ch3</code>	128
7.15.2.5 <code>warn_alert_ch2</code>	128
7.15.2.6 <code>warn_alert_ch1</code>	128
7.15.2.7 <code>shunt_sum_alert</code>	128
7.15.2.8 <code>crit_alert_ch3</code>	128
7.15.2.9 <code>crit_alert_ch2</code>	128
7.15.2.10 <code>crit_alert_ch1</code>	128
7.15.2.11 <code>crit_alert_latch_en</code>	128
7.15.2.12 <code>warn_alert_latch_en</code>	129
7.15.2.13 <code>shunt_sum_en_ch3</code>	129
7.15.2.14 <code>shunt_sum_en_ch2</code>	129
7.15.2.15 <code>shunt_sum_en_ch1</code>	129
7.15.2.16 <code>reserved</code>	129
7.16 <code>NMEADData</code> Class Reference	129
7.16.1 Detailed Description	130
7.16.2 Constructor & Destructor Documentation	131
7.16.2.1 <code>NMEADData()</code> [1/2]	131
7.16.2.2 <code>NMEADData()</code> [2/2]	131
7.16.3 Member Function Documentation	131
7.16.3.1 <code>operator=()</code>	131
7.16.3.2 <code>get_instance()</code>	131
7.16.3.3 <code>update_rmc_tokens()</code>	131
7.16.3.4 <code>update_gga_tokens()</code>	132
7.16.3.5 <code>get_rmc_tokens()</code>	132
7.16.3.6 <code>get_gga_tokens()</code>	132
7.16.3.7 <code>has_valid_time()</code>	133
7.16.3.8 <code>get_unix_time()</code>	133
7.16.4 Member Data Documentation	133
7.16.4.1 <code>rmc_tokens_</code>	133
7.16.4.2 <code>gga_tokens_</code>	133
7.16.4.3 <code>rmc_mutex_</code>	133
7.16.4.4 <code>gga_mutex_</code>	134
7.17 <code>PowerManager</code> Class Reference	134
7.17.1 Detailed Description	135
7.17.2 Constructor & Destructor Documentation	135

7.17.2.1 PowerManager() [1/2]	135
7.17.2.2 PowerManager() [2/2]	136
7.17.3 Member Function Documentation	136
7.17.3.1 operator=()	136
7.17.4 Member Data Documentation	136
7.17.4.1 SOLAR_CURRENT_THRESHOLD	136
7.17.4.2 USB_CURRENT_THRESHOLD	136
7.17.4.3 BATTERY_LOW_THRESHOLD	136
7.17.4.4 BATTERY_FULL_THRESHOLD	137
7.17.4.5 ina3221_	137
7.17.4.6 initialized_	137
7.17.4.7 powerman_mutex_	137
7.17.4.8 charging_solar_active_	137
7.17.4.9 charging_usb_active_	137
7.18 SensorDataRecord Struct Reference	138
7.18.1 Detailed Description	138
7.18.2 Member Data Documentation	138
7.18.2.1 timestamp	138
7.18.2.2 temperature	138
7.18.2.3 pressure	139
7.18.2.4 humidity	139
7.18.2.5 light	139
7.19 SensorWrapper Class Reference	139
7.19.1 Detailed Description	140
7.19.2 Constructor & Destructor Documentation	140
7.19.2.1 SensorWrapper()	140
7.19.3 Member Function Documentation	140
7.19.3.1 get_instance()	140
7.19.4 Member Data Documentation	141
7.19.4.1 sensors	141
7.20 SystemStateManager Class Reference	141
7.20.1 Detailed Description	142
7.20.2 Constructor & Destructor Documentation	143
7.20.2.1 SystemStateManager() [1/2]	143
7.20.2.2 SystemStateManager() [2/2]	143
7.20.3 Member Function Documentation	143
7.20.3.1 operator=()	143
7.20.3.2 get_instance()	143
7.20.3.3 is_bootloader_reset_pending()	143
7.20.3.4 set_bootloader_reset_pending()	143
7.20.3.5 is_gps_collection_paused()	144
7.20.3.6 set_gps_collection_paused()	144

7.20.3.7 is_sd_card_mounted()	144
7.20.3.8 set_sd_card_mounted()	144
7.20.3.9 get_uart_verbosity()	145
7.20.3.10 set_uart_verbosity()	145
7.20.3.11 is_radio_init_ok()	145
7.20.3.12 set_radio_init_ok()	145
7.20.3.13 is_light_sensor_init_ok()	146
7.20.3.14 set_light_sensor_init_ok()	146
7.20.3.15 is_env_sensor_init_ok()	146
7.20.3.16 set_env_sensor_init_ok()	146
7.20.4 Member Data Documentation	147
7.20.4.1 pending_bootloader_reset	147
7.20.4.2 gps_collection_paused	147
7.20.4.3 sd_card_mounted	147
7.20.4.4 uart_verbosity	147
7.20.4.5 sd_card_init_status	147
7.20.4.6 radio_init_status	148
7.20.4.7 light_sensor_init_status	148
7.20.4.8 env_sensor_init_status	148
7.20.4.9 mutex_	148
7.21 TelemetryManager Class Reference	148
7.21.1 Detailed Description	150
7.21.2 Constructor & Destructor Documentation	150
7.21.2.1 ~TelemetryManager()	150
7.21.3 Member Function Documentation	150
7.21.3.1 get_instance()	150
7.21.3.2 flush_sensor_data()	151
7.21.3.3 get_last_telemetry_record()	151
7.21.3.4 get_last_sensor_record()	151
7.21.3.5 get_telemetry_buffer_count()	151
7.21.3.6 get_telemetry_buffer_write_index()	151
7.21.4 Member Data Documentation	151
7.21.4.1 TELEMETRY_BUFFER_SIZE	151
7.21.4.2 DEFAULT_SAMPLE_INTERVAL_MS	152
7.21.4.3 DEFAULT_FLUSH_THRESHOLD	152
7.21.4.4 sample_interval_ms	152
7.21.4.5 flush_threshold	152
7.21.4.6 telemetry_buffer	152
7.21.4.7 telemetry_buffer_count	152
7.21.4.8 telemetry_buffer_write_index	152
7.21.4.9 sensor_data_buffer	153
7.21.4.10 telemetry_mutex	153

7.22 TelemetryRecord Struct Reference	153
7.22.1 Detailed Description	154
7.22.2 Member Data Documentation	154
7.22.2.1 timestamp	154
7.22.2.2 build_version	154
7.22.2.3 battery_voltage	154
7.22.2.4 system_voltage	154
7.22.2.5 charge_current_usb	154
7.22.2.6 charge_current_solar	155
7.22.2.7 discharge_current	155
7.22.2.8 time	155
7.22.2.9 latitude	155
7.22.2.10 lat_dir	155
7.22.2.11 longitude	155
7.22.2.12 lon_dir	156
7.22.2.13 speed	156
7.22.2.14 course	156
7.22.2.15 date	156
7.22.2.16 fix_quality	156
7.22.2.17 satellites	156
7.22.2.18 altitude	156
8 File Documentation	157
8.1 build_number.h File Reference	157
8.1.1 Macro Definition Documentation	157
8.1.1.1 BUILD_NUMBER	157
8.2 build_number.h	157
8.3 includes.h File Reference	158
8.4 includes.h	158
8.5 lib/clock/DS3231.cpp File Reference	159
8.6 DS3231.cpp	159
8.7 lib/clock/DS3231.h File Reference	163
8.7.1 Macro Definition Documentation	164
8.7.1.1 DS3231_DEVICE_ADRESS	164
8.7.1.2 DS3231_SECONDS_REG	164
8.7.1.3 DS3231_MINUTES_REG	165
8.7.1.4 DS3231_HOURS_REG	165
8.7.1.5 DS3231_DAY_REG	165
8.7.1.6 DS3231_DATE_REG	165
8.7.1.7 DS3231_MONTH_REG	165
8.7.1.8 DS3231_YEAR_REG	165
8.7.1.9 DS3231_CONTROL_REG	166

8.7.1.10 DS3231_CONTROL_STATUS_REG	166
8.7.1.11 DS3231_TEMPERATURE_MSB_REG	166
8.7.1.12 DS3231_TEMPERATURE_LSB_REG	166
8.7.2 Enumeration Type Documentation	166
8.7.2.1 days_of_week	166
8.8 DS3231.h	167
8.9 lib/comms/commands/clock_commands.cpp File Reference	168
8.9.1 Macro Definition Documentation	169
8.9.1.1 CLOCK_GROUP	169
8.9.1.2 TIME	169
8.9.1.3 TIMEZONE_OFFSET	169
8.9.1.4 CLOCK_SYNC_INTERVAL	169
8.9.1.5 LAST_SYNC_TIME	169
8.10 clock_commands.cpp	170
8.11 lib/comms/commands/commands.cpp File Reference	172
8.12 commands.cpp	173
8.13 lib/comms/commands/commands.h File Reference	174
8.14 commands.h	175
8.15 lib/comms/commands/diagnostic_commands.cpp File Reference	176
8.16 diagnostic_commands.cpp	177
8.17 lib/comms/commands/event_commands.cpp File Reference	178
8.18 event_commands.cpp	178
8.19 lib/comms/commands/gps_commands.cpp File Reference	179
8.19.1 Macro Definition Documentation	180
8.19.1.1 GPS_GROUP	180
8.19.1.2 POWER_STATUS_COMMAND	180
8.19.1.3 PASSTHROUGH_COMMAND	180
8.19.1.4 RMC_DATA_COMMAND	180
8.19.1.5 GGA_DATA_COMMAND	181
8.20 gps_commands.cpp	181
8.21 lib/comms/commands/power_commands.cpp File Reference	183
8.21.1 Macro Definition Documentation	184
8.21.1.1 POWER_GROUP	184
8.21.1.2 POWER_MANAGER_IDS	184
8.21.1.3 VOLTAGE_BATTERY	184
8.21.1.4 VOLTAGE_MAIN	184
8.21.1.5 CHARGE_USB	185
8.21.1.6 CHARGE_SOLAR	185
8.21.1.7 CHARGE_TOTAL	185
8.21.1.8 DRAW_TOTAL	185
8.22 power_commands.cpp	185
8.23 lib/comms/commands/sensor_commands.cpp File Reference	187

8.23.1 Macro Definition Documentation	188
8.23.1.1 SENSOR_GROUP	188
8.23.1.2 SENSOR_READ	188
8.23.1.3 SENSOR_CONFIGURE	188
8.24 sensor_commands.cpp	188
8.25 lib/comms/commands/storage_commands.cpp File Reference	191
8.25.1 Macro Definition Documentation	192
8.25.1.1 STORAGE_GROUP	192
8.25.1.2 LIST_FILES_COMMAND	192
8.25.1.3 MOUNT_COMMAND	192
8.26 storage_commands.cpp	192
8.27 lib/comms/commands/telemetry_commands.cpp File Reference	194
8.27.1 Macro Definition Documentation	194
8.27.1.1 TELEMETRY_GROUP	194
8.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND	195
8.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND	195
8.28 telemetry_commands.cpp	195
8.29 lib/comms/communication.cpp File Reference	196
8.29.1 Function Documentation	196
8.29.1.1 initialize_radio()	196
8.29.1.2 lora_tx_done_callback()	196
8.29.2 Variable Documentation	197
8.29.2.1 outgoing	197
8.29.2.2 msgCount	197
8.29.2.3 lastSendTime	197
8.29.2.4 lastReceiveTime	197
8.29.2.5 lastPrintTime	197
8.29.2.6 interval	197
8.30 communication.cpp	198
8.31 lib/comms/communication.h File Reference	198
8.31.1 Function Documentation	199
8.31.1.1 initialize_radio()	199
8.31.1.2 lora_tx_done_callback()	199
8.31.1.3 send_message()	199
8.31.1.4 send_frame_uart()	200
8.31.1.5 send_frame_lora()	200
8.31.1.6 split_and_send_message()	200
8.31.1.7 determine_unit()	200
8.32 communication.h	200
8.33 lib/comms/frame.cpp File Reference	201
8.33.1 Detailed Description	201
8.33.2 Typedef Documentation	201

8.33.2.1 CommandHandler	201
8.33.3 Variable Documentation	201
8.33.3.1 eventRegister	201
8.34 frame.cpp	202
8.35 lib/comms/protocol.h File Reference	203
8.35.1 Variable Documentation	205
8.35.1.1 FRAME_BEGIN	205
8.35.1.2 FRAME_END	205
8.35.1.3 DELIMITER	205
8.36 protocol.h	205
8.37 lib/comms/receive.cpp File Reference	207
8.37.1 Detailed Description	207
8.37.2 Macro Definition Documentation	207
8.37.2.1 MAX_PACKET_SIZE	207
8.38 receive.cpp	208
8.39 lib/comms/send.cpp File Reference	209
8.39.1 Detailed Description	209
8.39.2 Function Documentation	209
8.39.2.1 send_message()	209
8.39.2.2 send_frame_lora()	209
8.39.2.3 send_frame_uart()	210
8.40 send.cpp	210
8.41 lib/comms/utils_converters.cpp File Reference	210
8.41.1 Detailed Description	211
8.42 utils_converters.cpp	211
8.43 lib/eventman/event_manager.cpp File Reference	212
8.43.1 Detailed Description	212
8.44 event_manager.cpp	213
8.45 lib/eventman/event_manager.h File Reference	214
8.45.1 Detailed Description	216
8.45.2 Variable Documentation	216
8.45.2.1 id	216
8.45.2.2 timestamp	216
8.45.2.3 group	216
8.45.2.4 event	216
8.46 event_manager.h	217
8.47 lib/location/gps_collector.cpp File Reference	218
8.47.1 Detailed Description	219
8.48 gps_collector.cpp	219
8.49 lib/location/gps_collector.h File Reference	220
8.49.1 Detailed Description	220
8.50 gps_collector.h	220

8.51 lib/location/NMEA/NMEA_data.h File Reference	221
8.51.1 Detailed Description	221
8.52 NMEA_data.h	221
8.53 lib/pin_config.h File Reference	222
8.53.1 Macro Definition Documentation	223
8.53.1.1 DEBUG_UART_PORT	223
8.53.1.2 DEBUG_UART_BAUD_RATE	223
8.53.1.3 DEBUG_UART_TX_PIN	224
8.53.1.4 DEBUG_UART_RX_PIN	224
8.53.1.5 MAIN_I2C_PORT	224
8.53.1.6 MAIN_I2C_SDA_PIN	224
8.53.1.7 MAIN_I2C_SCL_PIN	224
8.53.1.8 GPS_UART_PORT	224
8.53.1.9 GPS_UART_BAUD_RATE	224
8.53.1.10 GPS_UART_TX_PIN	224
8.53.1.11 GPS_UART_RX_PIN	225
8.53.1.12 GPS_POWER_ENABLE_PIN	225
8.53.1.13 SENSORS_POWER_ENABLE_PIN	225
8.53.1.14 SENSORS_I2C_PORT	225
8.53.1.15 SENSORS_I2C_SDA_PIN	225
8.53.1.16 SENSORS_I2C_SCL_PIN	225
8.53.1.17 BUFFER_SIZE	225
8.53.1.18 SD_SPI_PORT	225
8.53.1.19 SD_MISO_PIN	226
8.53.1.20 SD_MOSI_PIN	226
8.53.1.21 SD_SCK_PIN	226
8.53.1.22 SD_CS_PIN	226
8.53.1.23 SD_CARD_DETECT_PIN	226
8.53.1.24 SX1278_MISO	226
8.53.1.25 SX1278_CS	226
8.53.1.26 SX1278_SCK	226
8.53.1.27 SX1278_MOSI	227
8.53.1.28 SPI_PORT	227
8.53.1.29 READ_BIT	227
8.53.1.30 LORA_DEFAULT_SPI	227
8.53.1.31 LORA_DEFAULT_SPI_FREQUENCY	227
8.53.1.32 LORA_DEFAULT_SS_PIN	227
8.53.1.33 LORA_DEFAULT_RESET_PIN	227
8.53.1.34 LORA_DEFAULT_DIO0_PIN	227
8.53.1.35 PA_OUTPUT_RFO_PIN	228
8.53.1.36 PA_OUTPUT_PA_BOOST_PIN	228
8.53.2 Variable Documentation	228

8.53.2.1 lora_cs_pin	228
8.53.2.2 lora_reset_pin	228
8.53.2.3 lora_irq_pin	228
8.53.2.4 lora_address_local	228
8.53.2.5 lora_address_remote	228
8.54 pin_config.h	229
8.55 lib/powerman/INA3221/INA3221.cpp File Reference	229
8.55.1 Detailed Description	230
8.56 INA3221.cpp	230
8.57 lib/powerman/INA3221/INA3221.h File Reference	233
8.57.1 Detailed Description	234
8.57.2 Enumeration Type Documentation	234
8.57.2.1 ina3221_addr_t	234
8.57.2.2 ina3221_ch_t	235
8.57.2.3 ina3221_reg_t	236
8.57.2.4 ina3221_conv_time_t	236
8.57.2.5 ina3221_avg_mode_t	237
8.57.3 Variable Documentation	237
8.57.3.1 INA3221_CH_NUM	237
8.57.3.2 SHUNT_VOLTAGE_LSB_UV	237
8.58 INA3221.h	238
8.59 lib/powerman/PowerManager.cpp File Reference	240
8.59.1 Detailed Description	240
8.60 PowerManager.cpp	240
8.61 lib/powerman/PowerManager.h File Reference	242
8.61.1 Detailed Description	242
8.62 PowerManager.h	243
8.63 lib/sensors/BH1750/BH1750.cpp File Reference	243
8.63.1 Detailed Description	244
8.64 BH1750.cpp	244
8.65 lib/sensors/BH1750/BH1750.h File Reference	244
8.65.1 Detailed Description	245
8.66 BH1750.h	245
8.67 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference	246
8.68 BH1750_WRAPPER.cpp	246
8.69 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference	247
8.70 BH1750_WRAPPER.h	247
8.71 lib/sensors/BME280/BME280.cpp File Reference	248
8.71.1 Detailed Description	248
8.72 BME280.cpp	248
8.73 lib/sensors/BME280/BME280.h File Reference	251
8.73.1 Detailed Description	251

8.74 BME280.h	251
8.75 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference	253
8.76 BME280_WRAPPER.cpp	253
8.77 lib/sensors/BME280/BME280_WRAPPER.h File Reference	254
8.78 BME280_WRAPPER.h	254
8.79 lib/sensors/ISensor.cpp File Reference	254
8.79.1 Detailed Description	254
8.80 ISensor.cpp	255
8.81 lib/sensors/ISensor.h File Reference	255
8.81.1 Detailed Description	256
8.82 ISensor.h	256
8.83 lib/storage/storage.cpp File Reference	257
8.83.1 Detailed Description	257
8.84 storage.cpp	258
8.85 lib/storage/storage.h File Reference	258
8.85.1 Detailed Description	259
8.86 storage.h	259
8.87 lib/system_state_manager.h File Reference	259
8.87.1 Detailed Description	260
8.88 system_state_manager.h	260
8.89 lib/telemetry/telemetry_manager.cpp File Reference	261
8.89.1 Detailed Description	262
8.90 telemetry_manager.cpp	262
8.91 lib/telemetry/telemetry_manager.h File Reference	266
8.91.1 Detailed Description	267
8.92 telemetry_manager.h	267
8.93 lib/utls.cpp File Reference	269
8.93.1 Detailed Description	269
8.93.2 Function Documentation	270
8.93.2.1 get_level_color()	270
8.93.2.2 get_level_prefix()	270
8.93.2.3 uart_print()	270
8.93.3 Variable Documentation	271
8.93.3.1 uart_mutex	271
8.94 utls.cpp	271
8.95 lib/utls.h File Reference	272
8.95.1 Detailed Description	272
8.95.2 Macro Definition Documentation	273
8.95.2.1 ANSI_RED	273
8.95.2.2 ANSI_GREEN	273
8.95.2.3 ANSI_YELLOW	273
8.95.2.4 ANSI_BLUE	273

8.95.2.5 ANSI_RESET	273
8.95.3 Enumeration Type Documentation	273
8.95.3.1 VerbosityLevel	273
8.95.4 Function Documentation	274
8.95.4.1 uart_print()	274
8.96 utils.h	274
8.97 main.cpp File Reference	275
8.97.1 Macro Definition Documentation	275
8.97.1.1 LOG_FILENAME	275
8.97.2 Function Documentation	275
8.97.2.1 core1_entry()	275
8.97.2.2 init_pico_hw()	275
8.97.2.3 init_modules()	276
8.97.2.4 main()	276
8.97.3 Variable Documentation	276
8.97.3.1 buffer	276
8.97.3.2 buffer_index	276
8.98 main.cpp	276
8.99 test/comms/commands/test_clock_commands.cpp File Reference	279
8.100 test_clock_commands.cpp	279
8.101 test/comms/commands/test_diagnostic_commands.cpp File Reference	279
8.101.1 Function Documentation	279
8.101.1.1 test_handle_get_commands_list()	279
8.101.1.2 test_handle_get_build_version()	279
8.101.1.3 test_handle_verbosity()	280
8.101.1.4 test_handle_enter_bootloader_mode()	280
8.102 test_diagnostic_commands.cpp	280
8.103 test/comms/commands/test_event_commands.cpp File Reference	281
8.104 test_event_commands.cpp	281
8.105 test/comms/commands/test_gps_commands.cpp File Reference	281
8.106 test_gps_commands.cpp	281
8.107 test/comms/commands/test_power_commands.cpp File Reference	281
8.108 test_power_commands.cpp	281
8.109 test/comms/commands/test_sensor_commands.cpp File Reference	281
8.110 test_sensor_commands.cpp	281
8.111 test/comms/commands/test_storage_commands.cpp File Reference	281
8.112 test_storage_commands.cpp	281
8.113 test/comms/commands/test_telemetry_commands.cpp File Reference	282
8.114 test_telemetry_commands.cpp	282
8.115 test/comms/test_comand_handlers.cpp File Reference	282
8.115.1 Function Documentation	282
8.115.1.1 send_frame_uart()	282

8.115.1.2 send_frame_lora()	282
8.115.1.3 setUp()	283
8.115.1.4 tearDown()	283
8.115.1.5 test_command_handler_get_operation()	283
8.115.1.6 test_command_handler_set_operation()	283
8.115.1.7 test_command_handler_invalid_operation()	283
8.115.2 Variable Documentation	283
8.115.2.1 uart_send_called	283
8.115.2.2 lora_send_called	283
8.115.2.3 last_frame_sent	284
8.116 test_comand_handlers.cpp	284
8.117 test/comms/test_converters.cpp File Reference	284
8.117.1 Function Documentation	285
8.117.1.1 test_operation_type_conversion()	285
8.117.1.2 test_value_unit_type_conversion()	285
8.117.1.3 test_exception_type_conversion()	285
8.117.1.4 test_hex_string_conversion()	285
8.118 test_converters.cpp	286
8.119 test/comms/test_frame_build.cpp File Reference	286
8.119.1 Function Documentation	286
8.119.1.1 test_frame_build_val()	286
8.119.1.2 test_frame_build_err()	287
8.119.1.3 test_frame_build_get()	287
8.119.1.4 test_frame_build_set()	287
8.119.1.5 test_frame_build_res()	287
8.119.1.6 test_frame_build_seq()	287
8.120 test_frame_build.cpp	288
8.121 test/comms/test_frame_coding.cpp File Reference	288
8.121.1 Function Documentation	289
8.121.1.1 test_frame_encode_basic()	289
8.121.1.2 test_frame_decode_basic()	289
8.121.1.3 test_frame_decode_invalid_header()	289
8.122 test_frame_coding.cpp	289
8.123 test/comms/test_frame_common.h File Reference	290
8.123.1 Function Documentation	290
8.123.1.1 create_test_frame()	290
8.124 test_frame_common.h	290
8.125 test/comms/test_frame_send.cpp File Reference	291
8.125.1 Function Documentation	291
8.125.1.1 setUp()	291
8.125.1.2 tearDown()	291
8.125.1.3 test_send_frame_uart()	291

8.126 test_frame_send.cpp	292
8.127 test/mocks/hardware_mocks.cpp File Reference	292
8.127.1 Function Documentation	292
8.127.1.1 mock_uart_puts()	292
8.127.1.2 mock_uart_init()	293
8.127.1.3 mock_spi_write_blocking()	293
8.127.1.4 mock_spi_read_blocking()	293
8.127.2 Variable Documentation	293
8.127.2.1 mock_uart_enabled	293
8.127.2.2 uart_output_buffer	293
8.127.2.3 mock_spi_enabled	293
8.127.2.4 spi_output_buffer	294
8.128 hardware_mocks.cpp	294
8.129 test/mocks/hardware_mocks.h File Reference	294
8.129.1 Function Documentation	295
8.129.1.1 mock_uart_puts()	295
8.129.1.2 mock_uart_init()	295
8.129.1.3 mock_spi_write_blocking()	295
8.129.1.4 mock_spi_read_blocking()	295
8.129.2 Variable Documentation	295
8.129.2.1 mock_uart_enabled	295
8.129.2.2 uart_output_buffer	296
8.129.2.3 mock_spi_enabled	296
8.129.2.4 spi_output_buffer	296
8.130 hardware_mocks.h	296
8.131 test/test_runner.cpp File Reference	296
8.131.1 Function Documentation	297
8.131.1.1 test_frame_encode_basic()	297
8.131.1.2 test_frame_decode_basic()	297
8.131.1.3 test_frame_decode_invalid_header()	297
8.131.1.4 test_frame_build_get()	298
8.131.1.5 test_frame_build_set()	298
8.131.1.6 test_frame_build_res()	298
8.131.1.7 test_frame_build_seq()	298
8.131.1.8 test_frame_build_val()	298
8.131.1.9 test_frame_build_err()	298
8.131.1.10 test_operation_type_conversion()	298
8.131.1.11 test_value_unit_type_conversion()	299
8.131.1.12 test_exception_type_conversion()	299
8.131.1.13 test_hex_string_conversion()	299
8.131.1.14 test_command_handler_get_operation()	299
8.131.1.15 test_command_handler_set_operation()	299

8.131.1.16 test_command_handler_invalid_operation()	299
8.131.1.17 test_handle_get_commands_list()	299
8.131.1.18 test_handle_get_build_version()	300
8.131.1.19 test_handle_verbosity()	300
8.131.1.20 test_handle_enter_bootloader_mode()	300
8.131.1.21 test_error_code_conversion()	300
8.131.1.22 main()	300
8.132 test_runner.cpp	300

Index	303
--------------	------------

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	11
Command System	13
Diagnostic Commands	16
Event Commands	18
GPS Commands	20
Power Commands	22
Sensor Commands	27
Storage Commands	29
Telemetry Buffer Commands	30
Frame Handling	31
Protocol	34
Receiving Data	36
Utility Converters	38
RTC clock	40
Event Management	51
Location	56
INA3221 Power Monitor	57
Configuration Functions	58
Measurement Functions	62
Power Management	63
BH1750 Light Sensor	68
Constants	70
Types	71
Sensors	72
Storage	76
System State Manager	77
Telemetry Manager	77

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

BH1750	85
BME280	90
BME280CalibParam	99
INA3221::conf_reg_t	106
DS3231	108
ds3231_data_t	112
EventEmitter	114
EventLog	115
EventManager	116
Frame	119
INA3221	121
ISensor	124
BH1750Wrapper	86
BME280Wrapper	103
INA3221::masken_reg_t	127
NMEADData	129
PowerManager	134
SensorDataRecord	138
SensorWrapper	139
SystemStateManager	141
TelemetryManager	148
TelemetryRecord	153

Chapter 4

Class Index

4.1 Class List

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

BH1750		
	Class to interface with the BH1750 light sensor	85
BH1750Wrapper		86
BME280		
	Class to interface with the BME280 environmental sensor	90
BME280CalibParam		
	Structure to hold the BME280 calibration parameters	99
BME280Wrapper		103
INA3221::conf_reg_t		
	Configuration register bit fields	106
DS3231		
	Class for interfacing with the DS3231 real-time clock	108
ds3231_data_t		
	Structure to hold time and date information from DS3231	112
EventEmitter		
	Provides a simple interface for emitting events	114
EventLog		
	Structure for storing event log data	115
EventManager		
	Manages event logging and storage	116
Frame		
	Represents a communication frame used for data exchange	119
INA3221		
	INA3221 Triple-Channel Power Monitor driver class	121
ISensor		
	Abstract base class for sensors	124
INA3221::masken_reg_t		
	Mask/Enable register bit fields	127
NMEAData		
	Manages parsed NMEA sentences	129
PowerManager		
	Manages power-related functions	134
SensorDataRecord		
	Structure representing a single sensor data point	138
SensorWrapper		
	Manages a collection of sensors	139

SystemStateManager	
Manages the system state of the Kubisat firmware	141
TelemetryManager	
Manages the collection, storage, and retrieval of telemetry data	148
TelemetryRecord	
Structure representing a single telemetry data point	153

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

build_number.h	157
includes.h	158
main.cpp	275
lib/pin_config.h	222
lib/system_state_manager.h	
Manages the system state of the Kubisat firmware	259
lib/utils.cpp	
Implementation of utility functions for the Kubisat firmware	269
lib/utils.h	
Utility functions and definitions for the Kubisat firmware	272
lib/clock/DS3231.cpp	159
lib/clock/DS3231.h	163
lib/comms/communication.cpp	196
lib/comms/communication.h	198
lib/comms/frame.cpp	
Implements functions for encoding, decoding, building, and processing Frames	201
lib/comms/protocol.h	203
lib/comms/receive.cpp	
Implements functions for receiving and processing data, including LoRa and UART input	207
lib/comms/send.cpp	
Implements functions for sending data, including LoRa messages and Frames	209
lib/comms/utils_converters.cpp	
Implements utility functions for converting between different data types	210
lib/comms/commands/clock_commands.cpp	168
lib/comms/commands/commands.cpp	172
lib/comms/commands/commands.h	174
lib/comms/commands/diagnostic_commands.cpp	176
lib/comms/commands/event_commands.cpp	178
lib/comms/commands/gps_commands.cpp	179
lib/comms/commands/power_commands.cpp	183
lib/comms/commands/sensor_commands.cpp	187
lib/comms/commands/storage_commands.cpp	191
lib/comms/commands/telemetry_commands.cpp	194
lib/eventman/event_manager.cpp	
Implementation of the Event Manager and Event Emitter classes	212

lib/eventman/event_manager.h	
Header file for the Event Manager and Event Emitter classes	214
lib/location/gps_collector.cpp	
Implementation of the GPS data collector module	218
lib/location/gps_collector.h	
Header file for the GPS data collector module	220
lib/location/NMEA/NMEA_data.h	
Header file for the NMEADData class, which manages parsed NMEA sentences	221
lib/powerman/PowerManager.cpp	
Implementation of the PowerManager class, which manages power-related functions	240
lib/powerman/PowerManager.h	
Header file for the PowerManager class, which manages power-related functions	242
lib/powerman/INA3221/INA3221.cpp	
Implementation of the INA3221 power monitor driver	229
lib/powerman/INA3221/INA3221.h	
Header file for the INA3221 triple-channel power monitor driver	233
lib/sensors/ISensor.cpp	
Implementation of the ISensor interface and SensorWrapper class	254
lib/sensors/ISensor.h	
Header file for the ISensor interface and SensorWrapper class	255
lib/sensors/BH1750/BH1750.cpp	
Implementation of the BH1750 light sensor class	243
lib/sensors/BH1750/BH1750.h	
Header file for the BH1750 light sensor class	244
lib/sensors/BH1750/BH1750_WRAPPER.cpp	246
lib/sensors/BH1750/BH1750_WRAPPER.h	247
lib/sensors/BME280/BME280.cpp	
Implementation of the BME280 environmental sensor class	248
lib/sensors/BME280/BME280.h	
Header file for the BME280 environmental sensor class	251
lib/sensors/BME280/BME280_WRAPPER.cpp	253
lib/sensors/BME280/BME280_WRAPPER.h	254
lib/storage/storage.cpp	
Implements file system operations for the Kubisat firmware	257
lib/storage/storage.h	
Header file for file system operations on the Kubisat firmware	258
lib/telemetry/telemetry_manager.cpp	
Implementation of telemetry collection and storage functionality	261
lib/telemetry/telemetry_manager.h	
System telemetry collection and logging	266
test/test_runner.cpp	296
test/comms/test_comand_handlers.cpp	282
test/comms/test_converters.cpp	284
test/comms/test_frame_build.cpp	286
test/comms/test_frame_coding.cpp	288
test/comms/test_frame_common.h	290
test/comms/test_frame_send.cpp	291
test/comms/commands/test_clock_commands.cpp	279
test/comms/commands/test_diagnostic_commands.cpp	279
test/comms/commands/test_event_commands.cpp	281
test/comms/commands/test_gps_commands.cpp	281
test/comms/commands/test_power_commands.cpp	281
test/comms/commands/test_sensor_commands.cpp	281
test/comms/commands/test_storage_commands.cpp	281
test/comms/commands/test_telemetry_commands.cpp	282
test/mocks/hardware_mock.cpp	292
test/mocks/hardware_mock.h	294

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.

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 31 of file [clock_commands.cpp](#).

6.1.2.2 handle_timezone_offset()

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

Handler for getting and setting timezone offset.

Parameters

<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 96 of file [clock_commands.cpp](#).

6.1.2.3 handle_clock_sync_interval()

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

Handler for getting and setting clock synchronization interval.

Parameters

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

Returns

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

Note

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

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

Command Command ID: 3.3

Definition at line 155 of file [clock_commands.cpp](#).

6.1.2.4 handle_get_last_sync_time()

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

Handler for getting last clock sync time.

Parameters

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

Returns

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

Note

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

Command Command ID: 3.7

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

6.2 Command System

Core command system implementation.

Typedefs

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

Functions

- `std::vector< Frame > execute_command` (`uint32_t commandKey`, `const std::string ¶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](#).

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](#).

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

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

Returns

One-element vector with result frame

Note

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

Get the firmware build version

Command Command ID: 1

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

6.3.2.3 handle_verbosity()

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

Handles setting or getting the UART verbosity level.

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

Parameters

<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](#).

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](#).

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

Note

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

Returns up to 10 most recent events per frame.

Command Command ID: 5.1

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

6.4.2.2 handle_get_event_count()

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

Handler for getting total number of events in the log.

Parameters

<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 101 of file [event_commands.cpp](#).

6.5 GPS Commands

Commands for controlling and monitoring the GPS module.

Functions

- `std::vector< Frame > handle_gps_power_status (const std::string ¶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

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

Returns

Vector of Frames containing:

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

Note

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

Return current GPS module power state: ON/OFF

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

POWER - 0 - OFF, 1 - ON

Command Command ID: 7.1

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

6.5.2.2 `handle_enable_gps_uart_passthrough()`

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

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

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

TIMEOUT - 1-600s, default 60s

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

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

Command Command ID: 7.2

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

6.5.2.3 `handle_get_rmc_data()`

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

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

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

Command Command ID: 7.3

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

6.5.2.4 `handle_get_gga_data()`

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

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

Parameters

<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 236 of file `gps_commands.cpp`.

6.6 Power Commands

Commands for monitoring power subsystem and battery management.

Functions

- `std::vector< Frame > handle_get_power_manager_ids` (const std::string ¶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](#).

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: 2.2

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

6.6.2.3 handle_get_voltage_5v()

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

Handler for getting 5V rail voltage.

Parameters

<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 96 of file [power_commands.cpp](#).

6.6.2.4 handle_get_current_charge_usb()

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

Handler for getting USB charge current.

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

This command is used to retrieve the USB charge current

Command Command ID: 2.4

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

6.6.2.5 `handle_get_current_charge_solar()`

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

Handler for getting solar panel charge current.

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

This command is used to retrieve the solar panel charge current

Command Command ID: 2.5

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

6.6.2.6 `handle_get_current_charge_total()`

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

Handler for getting total charge current.

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

This command is used to retrieve the total charge current

Command Command ID: 2.6

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

6.6.2.7 handle_get_current_draw()

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

Handler for getting system current draw.

Parameters

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

Returns

Vector of Frames containing:

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

Note

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

This command is used to retrieve the system current draw

Command Command ID: 2.7

Definition at line 228 of file [power_commands.cpp](#).

6.7 Sensor Commands

Commands for reading and configuring sensors.

Functions

- `std::vector< Frame > handle_get_sensor_data (const std::string ¶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"
<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 34 of file [sensor_commands.cpp](#).

6.7.2.2 handle_sensor_config()

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

Handler for configuring sensors.

Parameters

<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 172 of file [sensor_commands.cpp](#).

6.7.2.3 handle_get_sensor_list()

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

Handler for listing available sensors.

Parameters

<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 252 of file [sensor_commands.cpp](#).

6.8 Storage Commands

Commands for interacting with the SD card storage.

Functions

- `std::vector< Frame > handle_list_files (const std::string ¶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](#).

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

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

Returns

A vector of Frames indicating the result of the operation.

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

Note

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

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

Command Command ID: 6.4

Definition at line 126 of file [storage_commands.cpp](#).

6.9 Telemetry Buffer Commands

Commands for interacting with the telemetry buffer.

Functions

- `std::vector< Frame > handle_get_last_telemetry_record` (const std::string ¶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 32 of file [telemetry_commands.cpp](#).

6.9.2.2 `handle_get_last_sensor_record()`

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

Handles the get last sensor record command.

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

Parameters

<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 65 of file [telemetry_commands.cpp](#).

6.10 Frame Handling

Functions for encoding, decoding and building communication frames.

Functions

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

6.10.1 Detailed Description

Functions for encoding, decoding and building communication frames.

6.10.2 Function Documentation

6.10.2.1 frame_encode()

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

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

Parameters

<i>frame</i>	The Frame instance to encode.
--------------	---

Returns

The [Frame](#) encoded as a string.

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

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

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

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

6.10.2.2 frame_decode()

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

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

Parameters

<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](#).

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

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

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

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

6.10.2.4 frame_build()

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

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

Parameters

<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](#).

6.11 Protocol

Definitions for the communication protocol used by the satellite.

Classes

- struct [Frame](#)
Represents a communication frame used for data exchange.

Enumerations

- enum class [ErrorCode](#) {
[ErrorCode::PARAM_UNNECESSARY](#) , [ErrorCode::PARAM_REQUIRED](#) , [ErrorCode::PARAM_INVALID](#) ,
[ErrorCode::INVALID_OPERATION](#) ,
[ErrorCode::NOT_ALLOWED](#) , [ErrorCode::INVALID_FORMAT](#) , [ErrorCode::INVALID_VALUE](#) , [ErrorCode::FAIL_TO_SET](#)
 ,
[ErrorCode::INTERNAL_FAIL_TO_READ](#) , [ErrorCode::UNKNOWN_ERROR](#) }
Standard error codes for command responses.
- enum class [OperationType](#) {
[OperationType::GET](#) , [OperationType::SET](#) , [OperationType::RES](#) , [OperationType::VAL](#) ,
[OperationType::SEQ](#) , [OperationType::ERR](#) }
Represents the type of operation being performed.
- enum class [CommandAccessLevel](#) { [CommandAccessLevel::NONE](#) , [CommandAccessLevel::READ_ONLY](#)
 , [CommandAccessLevel::WRITE_ONLY](#) , [CommandAccessLevel::READ_WRITE](#) }
Represents the access level required to execute a command.
- enum class [ValueUnit](#) {
[ValueUnit::UNDEFINED](#) , [ValueUnit::SECOND](#) , [ValueUnit::VOLT](#) , [ValueUnit::BOOL](#) ,
[ValueUnit::DATETIME](#) , [ValueUnit::TEXT](#) , [ValueUnit::MILIAMP](#) }
Represents the unit of measurement for a payload value.
- enum class [ExceptionType](#) {
[ExceptionType::NONE](#) , [ExceptionType::NOT_ALLOWED](#) , [ExceptionType::INVALID_PARAM](#) , [ExceptionType::INVALID_OPERATION](#)
 ,
[ExceptionType::PARAM_UNNECESSARY](#) }
Represents the type of exception that occurred during command execution.
- enum class [Interface](#) { [Interface::UART](#) , [Interface::LORA](#) }
Represents the communication interface being used.

6.11.1 Detailed Description

Definitions for the communication protocol used by the satellite.

6.11.2 Enumeration Type Documentation

6.11.2.1 ErrorCode

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

Standard error codes for command responses.

Enumerator

PARAM_UNNECESSARY	
PARAM_REQUIRED	
PARAM_INVALID	
INVALID_OPERATION	
NOT_ALLOWED	
INVALID_FORMAT	
INVALID_VALUE	
FAIL_TO_SET	
INTERNAL_FAIL_TO_READ	
UNKNOWN_ERROR	

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

6.11.2.2 OperationType

```
enum class OperationType [strong]
```

Represents the type of operation being performed.

Enumerator

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

Definition at line 72 of file [protocol.h](#).

6.11.2.3 CommandAccessLevel

```
enum class CommandAccessLevel [strong]
```

Represents the access level required to execute a command.

Enumerator

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

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

6.11.2.4 ValueUnit

```
enum class ValueUnit [strong]
```

Represents the unit of measurement for a payload value.

Enumerator

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

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

6.11.2.5 ExceptionType

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

Represents the type of exception that occurred during command execution.

Enumerator

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

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

6.11.2.6 Interface

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

Represents the communication interface being used.

Enumerator

UART	UART interface.
LORA	LoRa interface.

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

6.12 Receiving Data

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

Functions

- void [on_receive](#) (int packet_size)
Callback function for handling received LoRa packets.
- void [handle_uart_input](#) ()
Handles UART input.

6.12.1 Detailed Description

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

6.12.2 Function Documentation

6.12.2.1 [on_receive\(\)](#)

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

Callback function for handling received LoRa packets.

Parameters

packet_size	The size of the received packet.
-----------------------------	----------------------------------

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

Definition at line 19 of file [receive.cpp](#).

6.12.2.2 [handle_uart_input\(\)](#)

```
void handle_uart_input ()
```

Handles UART input.

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

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

6.13 Utility Converters

Functions

- `std::string exception_type_to_string (ExceptionType type)`
Converts an *ExceptionType* to a string.
- `std::string value_unit_type_to_string (ValueUnit unit)`
Converts a *ValueUnit* to a string.
- `std::string operation_type_to_string (OperationType type)`
Converts an *OperationType* to a string.
- `OperationType string_to_operation_type (const std::string &str)`
Converts a string to an *OperationType*.
- `std::string error_code_to_string (ErrorCode code)`
Converts an *ErrorCode* to its string representation.
- `std::vector< uint8_t > hex_string_to_bytes (const std::string &hexString)`
Converts a hex string to a vector of bytes.

6.13.1 Detailed Description

6.13.2 Function Documentation

6.13.2.1 exception_type_to_string()

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

Converts an *ExceptionType* to a string.

Parameters

<i>type</i>	The <i>ExceptionType</i> to convert.
-------------	--------------------------------------

Returns

The string representation of the *ExceptionType*.

Definition at line 16 of file *utils_converters.cpp*.

6.13.2.2 value_unit_type_to_string()

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

Converts a *ValueUnit* to a string.

Parameters

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

Returns

The string representation of the *ValueUnit*.

Definition at line 34 of file *utils_converters.cpp*.

6.13.2.3 operation_type_to_string()

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

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

Parameters

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

Returns

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

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

6.13.2.4 string_to_operation_type()

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

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

Parameters

<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 73 of file [utils_converters.cpp](#).

6.13.2.5 error_code_to_string()

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

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

Parameters

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

Returns

String representation of the error code

Definition at line 89 of file [utils_converters.cpp](#).

6.13.2.6 hex_string_to_bytes()

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

Converts a hex string to a vector of bytes.

Parameters

<i>hexString</i>	The hex string to convert.
------------------	----------------------------

Returns

A vector of bytes representing the hex string.

Definition at line 111 of file [utils_converters.cpp](#).

6.14 RTC clock

Functions for interfacing with the [DS3231](#) RTC module.

Functions

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

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

6.14.1 Detailed Description

Functions for interfacing with the DS3231 RTC module.

6.14.2 Function Documentation

6.14.2.1 DS3231()

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

Constructor for the DS3231 class.

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

Note

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

Definition at line 23 of file DS3231.cpp.

6.14.2.2 get_instance()

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

Gets the singleton instance of the DS3231 class.

Returns

A reference to the singleton instance of the DS3231 class

A reference to the singleton instance of the DS3231 class

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

Definition at line 38 of file DS3231.cpp.

6.14.2.3 set_time()

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

Sets the time on the DS3231 clock.

Sets the time on the DS3231 RTC module.

Parameters

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

Returns

0 on success, -1 on failure

Parameters

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

Returns

0 on success, -1 on failure

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

Note

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

Definition at line [56](#) of file [DS3231.cpp](#).

6.14.2.4 get_time()

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

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

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

Parameters

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

Returns

0 on success, -1 on failure

Parameters

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

Returns

0 on success, -1 on failure

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

Note

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

Definition at line [126](#) of file [DS3231.cpp](#).

6.14.2.5 read_temperature()

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

Reads the current temperature from the [DS3231](#).

Reads the temperature from the [DS3231](#)'s internal temperature sensor.

Parameters

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

Returns

0 on success, -1 on failure

Parameters

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

Returns

0 on success, -1 on failure

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

Definition at line [168](#) of file [DS3231.cpp](#).

6.14.2.6 set_unix_time()

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

Sets the time using a Unix timestamp.

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

Parameters

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

Returns

0 on success, -1 on failure

Parameters

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

Returns

0 on success, -1 on failure

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

Definition at line 198 of file [DS3231.cpp](#).

6.14.2.7 get_unix_time()

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

Gets the current time as a Unix timestamp.

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

Returns

Unix timestamp, or -1 on error

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

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

Definition at line 229 of file [DS3231.cpp](#).

6.14.2.8 clock_enable()

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

Enables the [DS3231](#) clock oscillator.

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

Returns

0 on success, -1 on failure

0 on success, -1 on failure

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

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

6.14.2.9 get_timezone_offset()

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

Gets the current timezone offset.

Gets the currently configured timezone offset.

Returns

Timezone offset in minutes (-720 to +720)

The timezone offset in minutes

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

Definition at line 300 of file [DS3231.cpp](#).

6.14.2.10 set_timezone_offset()

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

Sets the timezone offset.

Parameters

	<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 316 of file [DS3231.cpp](#).

6.14.2.11 get_clock_sync_interval()

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

Gets the clock synchronization interval.

Gets the currently configured clock synchronization interval.

Returns

Sync interval in minutes

The sync interval in minutes

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

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

6.14.2.12 set_clock_sync_interval()

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

Sets the clock synchronization interval.

Parameters

	<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 [349](#) of file [DS3231.cpp](#).

6.14.2.13 get_last_sync_time()

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

Gets the timestamp of the last clock synchronization.

Gets the timestamp of the last successful clock synchronization.

Returns

Unix timestamp of last sync, 0 if never synced

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

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

Definition at line [367](#) of file [DS3231.cpp](#).

6.14.2.14 update_last_sync_time()

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

Updates the last sync time to current time.

Updates the last sync timestamp to the current time.

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

Definition at line [381](#) of file [DS3231.cpp](#).

6.14.2.15 get_local_time()

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

Gets the current local time (including timezone offset)

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

Returns

Unix timestamp adjusted for timezone, or -1 on error

Local time as Unix timestamp, or -1 on error

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

Definition at line 395 of file [DS3231.cpp](#).

6.14.2.16 is_sync_needed()

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

Checks if clock synchronization is needed.

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

Returns

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

true if synchronization is needed, false otherwise

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

Definition at line 415 of file [DS3231.cpp](#).

6.14.2.17 sync_clock_with_gps()

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

Synchronizes clock with GPS data.

Synchronizes the RTC with time from GPS.

Returns

true if sync successful, false otherwise

Parameters

in	<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 446 of file [DS3231.cpp](#).

6.14.2.18 i2c_read_reg()

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

Reads data from a specific register on the [DS3231](#).

Parameters

in	<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 493 of file [DS3231.cpp](#).

6.14.2.19 i2c_write_reg()

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

Writes data to a specific register on the [DS3231](#).

Parameters

in	<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 [535](#) of file [DS3231.cpp](#).

6.14.2.20 bin_to_bcd()

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

Converts binary value to BCD (Binary Coded Decimal)

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

Parameters

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

Returns

BCD representation of the input value

Parameters

<i>in</i>	<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 [566](#) of file [DS3231.cpp](#).

6.14.2.21 bcd_to_bin()

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

Converts BCD (Binary Coded Decimal) to binary value.

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

Parameters

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

Returns

Binary representation of the input BCD value

Parameters

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

Returns

Binary representation of the input BCD value

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

Definition at line [583](#) of file [DS3231.cpp](#).

6.15 Event Management

Classes and enums for handling system events.

Classes

- class [EventLog](#)
Structure for storing event log data.
- class [EventManager](#)
Manages event logging and storage.
- class [EventEmitter](#)
Provides a simple interface for emitting events.

Macros

- `#define EVENT_BUFFER_SIZE 100`
Size of the event buffer.
- `#define EVENT_FLUSH_THRESHOLD 10`
Number of events to accumulate before flushing to storage.
- `#define EVENT_LOG_FILE "/event_log.csv"`
Path to the event log file.

Enumerations

- enum class [EventGroup](#) : `uint8_t` {
 [EventGroup::SYSTEM](#) = 0x00 , [EventGroup::POWER](#) = 0x01 , [EventGroup::COMMS](#) = 0x02 ,
 [EventGroup::GPS](#) = 0x03 ,
 [EventGroup::CLOCK](#) = 0x04 }
Enumeration of event groups.
- enum class [SystemEvent](#) : `uint8_t` {
 [SystemEvent::BOOT](#) = 0x01 , [SystemEvent::SHUTDOWN](#) = 0x02 , [SystemEvent::WATCHDOG_RESET](#) =
 0x03 , [SystemEvent::CORE1_START](#) = 0x04 ,
 [SystemEvent::CORE1_STOP](#) = 0x05 }
Enumeration of system events.
- enum class [PowerEvent](#) : `uint8_t` {
 [PowerEvent::BATTERY_LOW](#) = 0x01 , [PowerEvent::BATTERY_FULL](#) = 0x02 , [PowerEvent::POWER_FALLING](#)
 = 0x03 , [PowerEvent::BATTERY_NORMAL](#) = 0x04 ,
 [PowerEvent::SOLAR_ACTIVE](#) = 0x05 , [PowerEvent::SOLAR_INACTIVE](#) = 0x06 , [PowerEvent::USB_CONNECTED](#)
 = 0x07 , [PowerEvent::USB_DISCONNECTED](#) = 0x08 }
Enumeration of power events.
- enum class [CommsEvent](#) : `uint8_t` {
 [CommsEvent::RADIO_INIT](#) = 0x01 , [CommsEvent::RADIO_ERROR](#) = 0x02 , [CommsEvent::MSG_RECEIVED](#)
 = 0x03 , [CommsEvent::MSG_SENT](#) = 0x04 ,
 [CommsEvent::UART_ERROR](#) = 0x06 }
Enumeration of communications events.
- enum class [GPSEvent](#) : `uint8_t` {
 [GPSEvent::LOCK](#) = 0x01 , [GPSEvent::LOST](#) = 0x02 , [GPSEvent::ERROR](#) = 0x03 , [GPSEvent::POWER_ON](#)
 = 0x04 ,
 [GPSEvent::POWER_OFF](#) = 0x05 , [GPSEvent::DATA_READY](#) = 0x06 , [GPSEvent::PASS_THROUGH_START](#)
 = 0x07 , [GPSEvent::PASS_THROUGH_END](#) = 0x08 }
Enumeration of GPS events.
- enum class [ClockEvent](#) : `uint8_t` { [ClockEvent::CHANGED](#) = 0x01 , [ClockEvent::GPS_SYNC](#) = 0x02 ,
 [ClockEvent::GPS_SYNC_DATA_NOT_READY](#) = 0x03 }
Enumeration of clock events.

Functions

- class `EventLog __attribute__((packed))`
- bool `EventManager::init ()`
Initializes the event manager.
- void `EventManager::log_event (uint8_t group, uint8_t event)`
Logs an event to the event buffer.
- const `EventLog & EventManager::get_event (size_t index) const`
Gets an event from the event buffer.
- bool `EventManager::save_to_storage ()`
Saves the event buffer to persistent storage.

Variables

- class `EventManager __attribute__((packed))`

6.15.1 Detailed Description

Classes and enums for handling system events.

6.15.2 Macro Definition Documentation

6.15.2.1 EVENT_BUFFER_SIZE

```
#define EVENT_BUFFER_SIZE 100
```

Size of the event buffer.

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

6.15.2.2 EVENT_FLUSH_THRESHOLD

```
#define EVENT_FLUSH_THRESHOLD 10
```

Number of events to accumulate before flushing to storage.

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

6.15.2.3 EVENT_LOG_FILE

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

Path to the event log file.

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

6.15.3 Enumeration Type Documentation

6.15.3.1 EventGroup

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

Enumeration of event groups.

Defines the different categories of events that can be logged.

Enumerator

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

Definition at line 50 of file [event_manager.h](#).

6.15.3.2 SystemEvent

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

Enumeration of system events.

Defines specific system-level events.

Enumerator

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

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

6.15.3.3 PowerEvent

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

Enumeration of power events.

Defines specific power management events.

Enumerator

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

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

6.15.3.4 CommsEvent

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

Enumeration of communications events.

Defines specific communications events.

Enumerator

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

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

6.15.3.5 GPSEvent

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

Enumeration of GPS events.

Defines specific GPS events.

Enumerator

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

Definition at line 130 of file [event_manager.h](#).

6.15.3.6 ClockEvent

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

Enumeration of clock events.

Defines specific clock-related events.

Enumerator

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

Definition at line 154 of file [event_manager.h](#).

6.15.4 Function Documentation

6.15.4.1 `__attribute__()`

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

6.15.4.2 `init()`

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

Initializes the event manager.

Returns

True if initialization was successful, false otherwise.

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

6.15.4.3 `log_event()`

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

Logs an event to the event buffer.

Parameters

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

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

6.15.4.4 `get_event()`

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

Gets an event from the event buffer.

Parameters

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

Returns

A const reference to the event log entry.

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

6.15.4.5 `save_to_storage()`

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

Saves the event buffer to persistent storage.

Returns

True if the save was successful, false otherwise.

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

6.15.5 Variable Documentation

6.15.5.1 `__attribute__`

```
class EventManager __attribute__
```

6.16 Location

Classes for handling location data.

Classes

- class [NMEAData](#)
Manages parsed NMEA sentences.

Macros

- `#define` [MAX_RAW_DATA_LENGTH](#) 256
Maximum length of the raw data buffer for NMEA sentences.

Functions

- `std::vector< std::string >` [splitString](#) (const std::string &str, char delimiter)
Splits a string into tokens based on a delimiter.
- void [collect_gps_data](#) ()
Collects GPS data from the UART and updates the NMEA data.

6.16.1 Detailed Description

Classes for handling location data.

6.16.2 Macro Definition Documentation

6.16.2.1 MAX_RAW_DATA_LENGTH

```
#define MAX_RAW_DATA_LENGTH 256
```

Maximum length of the raw data buffer for NMEA sentences.

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

6.16.3 Function Documentation

6.16.3.1 splitString()

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

Splits a string into tokens based on a delimiter.

Parameters

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

Returns

A vector of strings representing the tokens.

Definition at line 40 of file [gps_collector.cpp](#).

6.16.3.2 collect_gps_data()

```
void collect_gps_data ()
```

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

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

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

6.17 INA3221 Power Monitor

Topics

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

6.17.1 Detailed Description

6.17.2 Configuration Functions

Functions

- `INA3221::INA3221 (ina3221_addr_t addr, i2c_inst_t *i2c)`
Constructor for `INA3221` class.
- `bool INA3221::begin ()`
Initialize the `INA3221` device.
- `void INA3221::reset ()`
Reset the `INA3221` to default settings.
- `uint16_t INA3221::get_manufacturer_id ()`
Get the manufacturer ID of the device.
- `uint16_t INA3221::get_die_id ()`
Get the die ID of the device.
- `uint16_t INA3221::read_register (ina3221_reg_t reg)`
Read a register from the device.
- `void INA3221::set_mode_power_down ()`
Set device to power-down mode.
- `void INA3221::set_mode_continuous ()`
Set device to continuous measurement mode.
- `void INA3221::set_mode_triggered ()`
Set device to triggered measurement mode.
- `void INA3221::set_shunt_measurement_enable ()`
Enable shunt voltage measurements.
- `void INA3221::set_shunt_measurement_disable ()`
Disable shunt voltage measurements.
- `void INA3221::set_bus_measurement_enable ()`
Enable bus voltage measurements.
- `void INA3221::set_bus_measurement_disable ()`
Disable bus voltage measurements.
- `void INA3221::set_averaging_mode (ina3221_avg_mode_t mode)`
Set the averaging mode for measurements.
- `void INA3221::set_bus_conversion_time (ina3221_conv_time_t convTime)`
Set bus voltage conversion time.
- `void INA3221::set_shunt_conversion_time (ina3221_conv_time_t convTime)`
Set shunt voltage conversion time.

6.17.2.1 Detailed Description

Functions for configuring the `INA3221` device

6.17.2.2 Function Documentation

6.17.2.2.1 `INA3221()`

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

Constructor for `INA3221` class.

Parameters

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

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

6.17.2.2.2 begin()

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

Initialize the [INA3221](#) device.

Returns

true if initialization successful, false otherwise

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

Definition at line 51 of file [INA3221.cpp](#).

6.17.2.2.3 reset()

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

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

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

Definition at line 84 of file [INA3221.cpp](#).

6.17.2.2.4 get_manufacturer_id()

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

Get the manufacturer ID of the device.

Returns

16-bit manufacturer ID (should be 0x5449)

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

6.17.2.2.5 get_die_id()

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

Get the die ID of the device.

Returns

16-bit die ID (should be 0x3220)

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

6.17.2.2.6 read_register()

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

Read a register from the device.

Parameters

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

Returns

16-bit value read from the register

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

6.17.2.2.7 `set_mode_power_down()`

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

Set device to power-down mode.

Disables bus voltage and continuous measurements

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

6.17.2.2.8 `set_mode_continuous()`

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

Set device to continuous measurement mode.

Enables continuous measurement of bus voltage and shunt voltage

Definition at line 152 of file [INA3221.cpp](#).

6.17.2.2.9 `set_mode_triggered()`

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

Set device to triggered measurement mode.

Disables continuous measurements, requiring manual triggers

Definition at line 166 of file [INA3221.cpp](#).

6.17.2.2.10 `set_shunt_measurement_enable()`

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

Enable shunt voltage measurements.

Definition at line 179 of file [INA3221.cpp](#).

6.17.2.2.11 set_shunt_measurement_disable()

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

Disable shunt voltage measurements.

Definition at line 192 of file [INA3221.cpp](#).

6.17.2.2.12 set_bus_measurement_enable()

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

Enable bus voltage measurements.

Definition at line 205 of file [INA3221.cpp](#).

6.17.2.2.13 set_bus_measurement_disable()

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

Disable bus voltage measurements.

Definition at line 218 of file [INA3221.cpp](#).

6.17.2.2.14 set_averaging_mode()

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

Set the averaging mode for measurements.

Parameters

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

Definition at line 232 of file [INA3221.cpp](#).

6.17.2.2.15 set_bus_conversion_time()

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

Set bus voltage conversion time.

Parameters

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

Definition at line 246 of file [INA3221.cpp](#).

6.17.2.2.16 set_shunt_conversion_time()

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

Set shunt voltage conversion time.

Parameters

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

Definition at line 260 of file [INA3221.cpp](#).

6.17.3 Measurement Functions

Functions

- `int32_t INA3221::get_shunt_voltage (ina3221_ch_t channel)`
Get shunt voltage for a specific channel.
- `float INA3221::get_current_ma (ina3221_ch_t channel)`
Get current for a specific channel.
- `float INA3221::get_voltage (ina3221_ch_t channel)`
Get bus voltage for a specific channel.

6.17.3.1 Detailed Description

Functions for reading voltage, current and power measurements

6.17.3.2 Function Documentation

6.17.3.2.1 `get_shunt_voltage()`

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

Get shunt voltage for a specific channel.

Parameters

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

Returns

Shunt voltage in microvolts (μV)

Definition at line 276 of file [INA3221.cpp](#).

6.17.3.2.2 `get_current_ma()`

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

Get current for a specific channel.

Parameters

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

Returns

Current in milliamps (mA)

Definition at line 308 of file [INA3221.cpp](#).

6.17.3.2.3 get_voltage()

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

Get bus voltage for a specific channel.

Parameters

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

Returns

Voltage in volts (V)

Definition at line 324 of file [INA3221.cpp](#).

6.18 Power Management

Classes for handling power-related functions.

Classes

- class [PowerManager](#)
Manages power-related functions.

Functions

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

6.18.1 Detailed Description

Classes for handling power-related functions.

6.18.2 Function Documentation

6.18.2.1 PowerManager()

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

Private constructor for the singleton pattern.

Initializes the [INA3221](#) and mutex.

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

6.18.2.2 `get_instance()`

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

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

Returns

A reference to the singleton instance.

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

6.18.2.3 `initialize()`

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

Initializes the [PowerManager](#).

Returns

True if initialization was successful, false otherwise.

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

6.18.2.4 `read_device_ids()`

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

Reads the manufacturer and die IDs from the [INA3221](#).

Returns

A string containing the manufacturer and die IDs.

Definition at line 58 of file [PowerManager.cpp](#).

6.18.2.5 `get_voltage_battery()`

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

Gets the battery voltage.

Returns

The battery voltage in volts.

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

6.18.2.6 `get_voltage_5v()`

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

Gets the 5V voltage.

Returns

The 5V voltage in volts.

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

6.18.2.7 `get_current_charge_usb()`

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

Gets the USB charging current.

Returns

The USB charging current in milliamperes.

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

6.18.2.8 `get_current_draw()`

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

Gets the current draw.

Returns

The current draw in milliamperes.

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

6.18.2.9 `get_current_charge_solar()`

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

Gets the solar charging current.

Returns

The solar charging current in milliamperes.

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

6.18.2.10 `get_current_charge_total()`

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

Gets the total charging current.

Returns

The total charging current in milliamperes.

Definition at line [142](#) of file [PowerManager.cpp](#).

6.18.2.11 `configure()`

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

Configures the [INA3221](#).

Parameters

in	config	A map of configuration parameters.
----	--------	------------------------------------

Definition at line 155 of file [PowerManager.cpp](#).

6.18.2.12 is_charging_solar()

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

Checks if solar charging is active.

Returns

True if solar charging is active, false otherwise.

Definition at line 189 of file [PowerManager.cpp](#).

6.18.2.13 is_charging_usb()

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

Checks if USB charging is active.

Returns

True if USB charging is active, false otherwise.

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

6.19 BH1750 Light Sensor

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

Topics

- [Constants](#)

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

- [Types](#)

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

Classes

- class [BH1750](#)

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

Functions

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

6.19.1 Detailed Description

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

6.19.2 Function Documentation

6.19.2.1 BH1750()

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

Constructor for the [BH1750](#) class.

Parameters

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

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

6.19.2.2 begin()

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

Initializes the [BH1750](#) sensor.

Parameters

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

Returns

True if initialization was successful, false otherwise.

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

6.19.2.3 configure()

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

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

Parameters

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

Returns

True if configuration was successful, false otherwise.

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

6.19.2.4 `get_light_level()`

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

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

Returns

Light level in lux.

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

6.19.2.5 `write8()`

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

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

Parameters

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

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

6.19.3 Constants

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

Macros

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

6.19.3.1 Detailed Description

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

6.19.3.2 Macro Definition Documentation

6.19.3.2.1 `_BH1750_DEVICE_ID`

```
#define _BH1750_DEVICE_ID 0xE1
```

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

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

6.19.3.2.2 `_BH1750_MTREG_MIN`

```
#define _BH1750_MTREG_MIN 31
```

Minimum value for the MTREG register.

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

6.19.3.2.3 `_BH1750_MTREG_MAX`

```
#define _BH1750_MTREG_MAX 254
```

Maximum value for the MTREG register.

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

6.19.3.2.4 `_BH1750_DEFAULT_MTREG`

```
#define _BH1750_DEFAULT_MTREG 69
```

Default value for the MTREG register.

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

6.19.4 Types

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

Enumerations

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

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

6.19.4.1 Detailed Description

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

6.19.4.2 Enumeration Type Documentation

6.19.4.2.1 Mode

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

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

Enumerator

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

Definition at line 66 of file [BH1750.h](#).

6.20 Sensors

Classes for handling sensor-related functions.

Classes

- class [ISensor](#)
Abstract base class for sensors.
- class [SensorWrapper](#)
Manages a collection of sensors.

Enumerations

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

Enumeration of sensor types.

- enum class `SensorDataTypeIdentifier` : `uint8_t` { `SensorDataTypeIdentifier::NONE` = 0x00 , `SensorDataTypeIdentifier::LIGHT_LEVEL` = 0x01 , `SensorDataTypeIdentifier::TEMPERATURE` = 0x02 , `SensorDataTypeIdentifier::HUMIDITY` = 0x03 , `SensorDataTypeIdentifier::PRESSURE` = 0x04 }

Enumeration of sensor data type identifiers.

Functions

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

6.20.1 Detailed Description

Classes for handling sensor-related functions.

6.20.2 Enumeration Type Documentation

6.20.2.1 SensorType

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

Enumeration of sensor types.

Defines the different types of sensors that can be managed.

Enumerator

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

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

6.20.2.2 SensorDataTypeIdentifier

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

Enumeration of sensor data type identifiers.

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

Enumerator

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

Definition at line 45 of file [ISensor.h](#).

6.20.3 Function Documentation**6.20.3.1 sensor_init()**

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

Initializes a sensor.

Parameters

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

Returns

True if initialization was successful, false otherwise.

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

6.20.3.2 sensor_configure()

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

Configures a sensor.

Parameters

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

Returns

True if configuration was successful, false otherwise.

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

6.20.3.3 sensor_read_data()

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

Reads data from a sensor.

Parameters

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

Returns

The sensor data.

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

6.20.3.4 get_sensor()

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

Gets a sensor.

Parameters

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

Returns

A pointer to the sensor.

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

6.20.3.5 scan_connected_sensors()

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

Scans for connected sensors.

Parameters

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

Returns

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

Definition at line 85 of file [ISensor.cpp](#).

6.20.3.6 `get_available_sensors()`

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

Gets a list of available sensors.

Returns

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

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

6.21 Storage

Classes and functions for managing file system operations.

Functions

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

6.21.1 Detailed Description

Classes and functions for managing file system operations.

6.21.2 Function Documentation

6.21.2.1 `fs_init()`

```
bool fs_init (  
    void )
```

Initializes the file system on the SD card.

Returns

True if initialization was successful, false otherwise.

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

Returns

True if initialization was successful, false otherwise.

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

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

6.21.2.2 fs_stop()

```
bool fs_stop (
    void )
```

Unmounts the file system from the SD card.

Returns

True if unmounting was successful, false otherwise.

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

6.22 System State Manager

Classes for handling system state management.

Classes

- class [SystemStateManager](#)
Manages the system state of the Kubisat firmware.

6.22.1 Detailed Description

Classes for handling system state management.

6.23 Telemetry Manager

Classes

- struct [TelemetryRecord](#)
Structure representing a single telemetry data point.
- struct [SensorDataRecord](#)
Structure representing a single sensor data point.
- class [TelemetryManager](#)
Manages the collection, storage, and retrieval of telemetry data.

Macros

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

Functions

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

6.23.1 Detailed Description

6.23.2 Macro Definition Documentation

6.23.2.1 TELEMETRY_CSV_PATH

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

Path to the telemetry CSV file on storage media.

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

6.23.2.2 SENSOR_DATA_CSV_PATH

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

Path to the sensor data CSV file on storage media.

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

6.23.2.3 DEFAULT_SAMPLE_INTERVAL_MS

```
#define DEFAULT_SAMPLE_INTERVAL_MS 1000
```

Default interval between telemetry samples in milliseconds (2 seconds)

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

6.23.2.4 DEFAULT_FLUSH_THRESHOLD

```
#define DEFAULT_FLUSH_THRESHOLD 10
```

Default number of records to collect before flushing to storage.

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

6.23.3 Function Documentation

6.23.3.1 to_csv() [1/2]

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

Converts the telemetry record to a CSV string.

Returns

A CSV string representing the telemetry record.

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

6.23.3.2 to_csv() [2/2]

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

Converts the sensor data record to a CSV string.

Returns

A CSV string representing the sensor data record.

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

6.23.3.3 collect_power_telemetry()

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

Collects power subsystem telemetry data.

Parameters

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

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

6.23.3.4 emit_power_events()

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

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

Parameters

in	<i>battery_voltage</i>	The current battery voltage.
in	<i>charge_current_usb</i>	The current USB charging current.
in	<i>charge_current_solar</i>	The current solar charging current.

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

6.23.3.5 collect_gps_telemetry()

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

Collects GPS telemetry data.

Parameters

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

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

6.23.3.6 collect_sensor_telemetry()

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

Collects sensor telemetry data.

Parameters

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

Definition at line 218 of file [telemetry_manager.cpp](#).

6.23.3.7 TelemetryManager()

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

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

6.23.3.8 init()

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

Initialize the telemetry system.

Initializes the telemetry manager.

Returns

True if initialization was successful

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

Returns

True if initialization was successful, false otherwise.

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

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

6.23.3.9 collect_telemetry()

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

Collect telemetry data from sensors and power subsystems.

Returns

True if data was successfully collected

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

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

6.23.3.10 flush_telemetry()

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

Save buffered telemetry data to storage.

Save buffered telemetry and sensor data to storage.

Returns

True if data was successfully saved

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

Returns

True if data was successfully saved

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

Definition at line 275 of file [telemetry_manager.cpp](#).

6.23.3.11 is_telemetry_collection_time()

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

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

Parameters

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

Returns

True if collection interval has passed

Updates last_collection_time if the interval has passed

Definition at line 331 of file [telemetry_manager.cpp](#).

6.23.3.12 is_telemetry_flush_time()

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

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

Parameters

<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 347 of file [telemetry_manager.cpp](#).

6.23.3.13 `get_last_telemetry_record_csv()`

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

Gets the last telemetry record as a CSV string.

Returns

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

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

6.23.3.14 `get_last_sensor_record_csv()`

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

Gets the last sensor data record as a CSV string.

Returns

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

Definition at line 380 of file [telemetry_manager.cpp](#).

Chapter 7

Class Documentation

7.1 BH1750 Class Reference

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

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

Public Types

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

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

Public Member Functions

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

Private Member Functions

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

Private Attributes

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

7.1.1 Detailed Description

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

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

7.1.2 Member Data Documentation

7.1.2.1 `_i2c_addr`

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

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

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

7.1.2.2 `i2c_port_`

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

Pointer to the I2C interface.

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

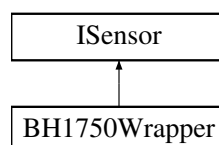
The documentation for this class was generated from the following files:

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

7.2 BH1750Wrapper Class Reference

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

Inheritance diagram for BH1750Wrapper:



Public Member Functions

- [BH1750Wrapper](#) (i2c_inst_t *i2c)
- [BH1750Wrapper](#) ()
- int [get_i2c_addr](#) ()
- bool [init](#) () override
Initializes the sensor.
- float [read_data](#) ([SensorDataTypeIdentifier](#) type) override
Reads data from the sensor.
- bool [is_initialized](#) () const override
Checks if the sensor is initialized.
- [SensorType](#) [get_type](#) () const override
Gets the sensor type.
- bool [configure](#) (const std::map< std::string, std::string > &config)
Configures the sensor.
- uint8_t [get_address](#) () const override
Gets the I2C address of the sensor.

Public Member Functions inherited from [ISensor](#)

- virtual [~ISensor](#) ()=default
Virtual destructor.

Private Attributes

- [BH1750](#) [sensor_](#)
- bool [initialized_](#) = false

7.2.1 Detailed Description

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

7.2.2 Constructor & Destructor Documentation

7.2.2.1 BH1750Wrapper() [1/2]

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

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

7.2.2.2 BH1750Wrapper() [2/2]

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

7.2.3 Member Function Documentation

7.2.3.1 `get_i2c_addr()`

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

7.2.3.2 `init()`

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

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implements [ISensor](#).

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

7.2.3.3 `read_data()`

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

Reads data from the sensor.

Parameters

in	type	Data type to read.
----	------	--------------------

Returns

The sensor data.

Implements [ISensor](#).

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

7.2.3.4 `is_initialized()`

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

Checks if the sensor is initialized.

Returns

True if the sensor is initialized, false otherwise.

Implements [ISensor](#).

Definition at line 21 of file [BH1750_WRAPPER.cpp](#).

7.2.3.5 get_type()

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

Gets the sensor type.

Returns

The sensor type.

Implements [ISensor](#).

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

7.2.3.6 configure()

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

Configures the sensor.

Parameters

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

Returns

True if configuration was successful, false otherwise.

Implements [ISensor](#).

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

7.2.3.7 get_address()

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

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

Implements [ISensor](#).

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

7.2.4 Member Data Documentation

7.2.4.1 sensor_

[BH1750](#) `BH1750Wrapper::sensor_` [private]

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

7.2.4.2 initialized_

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

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

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

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

7.3 BME280 Class Reference

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

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

Public Types

- enum { [ADDR_SDO_LOW](#) = 0x76 , [ADDR_SDO_HIGH](#) = 0x77 }
I2C Address Options for the [BME280](#) sensor.
- enum class [Oversampling](#) : uint8_t {
 [OSR_X0](#) = 0x00 , [OSR_X1](#) = 0x01 , [OSR_X2](#) = 0x02 , [OSR_X4](#) = 0x03 ,
 [OSR_X8](#) = 0x04 , [OSR_X16](#) = 0x05 }
Enum class for oversampling settings.

Public Member Functions

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

Private Types

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

Register Definitions for the BME280 sensor.

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

Sensor settings.

- enum { [NUM_CALIB_PARAMS](#) = 26 , [NUM_HUM_CALIB_PARAMS](#) = 7 }

Calibration data length.

Private Member Functions

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

Private Attributes

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

7.3.1 Detailed Description

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

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

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

7.3.2 Member Enumeration Documentation

7.3.2.1 anonymous enum

anonymous enum

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

Enumerator

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

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

7.3.2.2 Oversampling

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

Enum class for oversampling settings.

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

Enumerator

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

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

7.3.2.3 anonymous enum

anonymous enum [private]

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

Enumerator

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

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

7.3.2.4 anonymous enum

```
anonymous enum [private]
```

Sensor settings.

Enumerator

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

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

7.3.2.5 anonymous enum

```
anonymous enum [private]
```

Calibration data length.

Enumerator

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

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

7.3.3 Constructor & Destructor Documentation

7.3.3.1 BME280()

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

Constructor for the [BME280](#) class.

Parameters

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

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

7.3.4 Member Function Documentation

7.3.4.1 init()

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

Initializes the [BME280](#) sensor.

Returns

True if initialization was successful, false otherwise.

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

7.3.4.2 reset()

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

Resets the [BME280](#) sensor.

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

7.3.4.3 read_raw_all()

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

Reads all raw data from the sensor.

Parameters

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

Returns

True if the data was read successfully, false otherwise.

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

7.3.4.4 convert_temperature()

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

Converts raw temperature data to degrees Celsius.

Parameters

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

Returns

Temperature in degrees Celsius.

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

7.3.4.5 convert_pressure()

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

Converts raw pressure data to hectopascals (hPa).

Parameters

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

Returns

Pressure in hPa.

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

7.3.4.6 convert_humidity()

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

Converts raw humidity data to relative humidity (%).

Parameters

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

Returns

Relative humidity in %.

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

7.3.4.7 write_register()

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

Helper function for I2C writes.

Parameters

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

Returns

True if the write was successful, false otherwise.

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

7.3.4.8 read_register() [1/2]

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

Helper function for I2C reads.

Parameters

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

Returns

True if the read was successful, false otherwise.

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

7.3.4.9 read_register() [2/2]

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

Helper function for I2C reads with a specified length.

Parameters

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

Returns

True if the read was successful, false otherwise.

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

7.3.4.10 configure_sensor()

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

Configures the sensor with default settings.

Returns

True if the configuration was successful, false otherwise.

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

7.3.4.11 `get_calibration_parameters()`

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

Retrieves the calibration parameters from the sensor.

Returns

True if the parameters were read successfully, false otherwise.

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

7.3.5 Member Data Documentation

7.3.5.1 `i2c_port`

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

Pointer to the I2C interface.

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

7.3.5.2 `device_addr`

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

I2C device address.

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

7.3.5.3 `calib_params`

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

Calibration parameters for the sensor.

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

7.3.5.4 `initialized_`

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

Initialization status of the sensor.

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

7.3.5.5 t_fine

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

Fine temperature parameter needed for compensation.

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

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

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

7.4 BME280CalibParam Struct Reference

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

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

Public Attributes

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

7.4.1 Detailed Description

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

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

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

7.4.2 Member Data Documentation

7.4.2.1 `dig_t1`

```
uint16_t BME280CalibParam::dig_t1
```

Temperature calibration parameter 1.

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

7.4.2.2 `dig_t2`

```
int16_t BME280CalibParam::dig_t2
```

Temperature calibration parameter 2.

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

7.4.2.3 `dig_t3`

```
int16_t BME280CalibParam::dig_t3
```

Temperature calibration parameter 3.

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

7.4.2.4 `dig_p1`

```
uint16_t BME280CalibParam::dig_p1
```

Pressure calibration parameter 1.

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

7.4.2.5 `dig_p2`

```
int16_t BME280CalibParam::dig_p2
```

Pressure calibration parameter 2.

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

7.4.2.6 dig_p3

```
int16_t BME280CalibParam::dig_p3
```

Pressure calibration parameter 3.

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

7.4.2.7 dig_p4

```
int16_t BME280CalibParam::dig_p4
```

Pressure calibration parameter 4.

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

7.4.2.8 dig_p5

```
int16_t BME280CalibParam::dig_p5
```

Pressure calibration parameter 5.

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

7.4.2.9 dig_p6

```
int16_t BME280CalibParam::dig_p6
```

Pressure calibration parameter 6.

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

7.4.2.10 dig_p7

```
int16_t BME280CalibParam::dig_p7
```

Pressure calibration parameter 7.

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

7.4.2.11 dig_p8

```
int16_t BME280CalibParam::dig_p8
```

Pressure calibration parameter 8.

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

7.4.2.12 dig_p9

```
int16_t BME280CalibParam::dig_p9
```

Pressure calibration parameter 9.

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

7.4.2.13 dig_h1

```
uint8_t BME280CalibParam::dig_h1
```

Humidity calibration parameter 1.

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

7.4.2.14 dig_h2

```
int16_t BME280CalibParam::dig_h2
```

Humidity calibration parameter 2.

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

7.4.2.15 dig_h3

```
uint8_t BME280CalibParam::dig_h3
```

Humidity calibration parameter 3.

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

7.4.2.16 dig_h4

```
int16_t BME280CalibParam::dig_h4
```

Humidity calibration parameter 4.

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

7.4.2.17 dig_h5

```
int16_t BME280CalibParam::dig_h5
```

Humidity calibration parameter 5.

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

7.4.2.18 dig_h6

```
int8_t BME280CalibParam::dig_h6
```

Humidity calibration parameter 6.

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

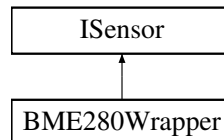
The documentation for this struct was generated from the following file:

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

7.5 BME280Wrapper Class Reference

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

Inheritance diagram for BME280Wrapper:



Public Member Functions

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

Public Member Functions inherited from [ISensor](#)

- virtual [~ISensor](#) ()=default
Virtual destructor.

Private Attributes

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

7.5.1 Detailed Description

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

7.5.2 Constructor & Destructor Documentation

7.5.2.1 BME280Wrapper()

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

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

7.5.3 Member Function Documentation

7.5.3.1 init()

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

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

Implements [ISensor](#).

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

7.5.3.2 read_data()

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

Reads data from the sensor.

Parameters

in	type	Data type to read.
----	------	--------------------

Returns

The sensor data.

Implements [ISensor](#).

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

7.5.3.3 is_initialized()

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

Checks if the sensor is initialized.

Returns

True if the sensor is initialized, false otherwise.

Implements [ISensor](#).

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

7.5.3.4 get_type()

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

Gets the sensor type.

Returns

The sensor type.

Implements [ISensor](#).

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

7.5.3.5 configure()

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

Configures the sensor.

Parameters

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

Returns

True if configuration was successful, false otherwise.

Implements [ISensor](#).

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

7.5.3.6 get_address()

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

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

Implements [ISensor](#).

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

7.5.4 Member Data Documentation

7.5.4.1 sensor_

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

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

7.5.4.2 initialized_

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

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

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

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

7.6 INA3221::conf_reg_t Struct Reference

Configuration register bit fields.

Public Attributes

- uint16_t [mode_shunt_en](#):1
- uint16_t [mode_bus_en](#):1
- uint16_t [mode_continuous_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_continuous_en

```
uint16_t INA3221::conf_reg_t::mode_continuous_en
```

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

7.6.2.4 shunt_conv_time

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

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

7.6.2.5 bus_conv_time

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

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

7.6.2.6 avg_mode

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

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

7.6.2.7 ch3_en

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

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

7.6.2.8 ch2_en

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

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

7.6.2.9 ch1_en

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

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

7.6.2.10 reset

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

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

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

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

7.7 DS3231 Class Reference

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

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

Public Member Functions

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

Static Public Member Functions

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

Private Member Functions

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

Private Attributes

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

7.7.1 Detailed Description

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

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

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

7.7.2 Constructor & Destructor Documentation

7.7.2.1 DS3231() [1/2]

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

Constructor for the [DS3231](#) class.

Parameters

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

7.7.2.2 DS3231() [2/2]

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

7.7.3 Member Function Documentation

7.7.3.1 operator=()

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


7.7.4 Member Data Documentation

7.7.4.1 i2c

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

Definition at line 232 of file [DS3231.h](#).

7.7.4.2 ds3231_addr

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

Definition at line 233 of file [DS3231.h](#).

7.7.4.3 clock_mutex_

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

Definition at line 234 of file [DS3231.h](#).

7.7.4.4 timezone_offset_minutes_

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

Definition at line 235 of file [DS3231.h](#).

7.7.4.5 sync_interval_minutes_

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

Definition at line 236 of file [DS3231.h](#).

7.7.4.6 last_sync_time_

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

Definition at line 237 of file [DS3231.h](#).

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

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

7.8 ds3231_data_t Struct Reference

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

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

Public Attributes

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

7.8.1 Detailed Description

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

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

7.8.2 Member Data Documentation

7.8.2.1 seconds

```
uint8_t ds3231_data_t::seconds
```

Seconds (0-59)

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

7.8.2.2 minutes

```
uint8_t ds3231_data_t::minutes
```

Minutes (0-59)

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

7.8.2.3 hours

```
uint8_t ds3231_data_t::hours
```

Hours (0-23)

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

7.8.2.4 day

```
uint8_t ds3231_data_t::day
```

Day of the week (1-7)

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

7.8.2.5 date

```
uint8_t ds3231_data_t::date
```

Date (1-31)

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

7.8.2.6 month

```
uint8_t ds3231_data_t::month
```

Month (1-12)

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

7.8.2.7 year

```
uint8_t ds3231_data_t::year
```

Year (0-99)

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

7.8.2.8 century

```
bool ds3231_data_t::century
```

Century flag (0-1)

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

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

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

7.9 EventEmitter Class Reference

Provides a simple interface for emitting events.

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

Static Public Member Functions

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

7.9.1 Detailed Description

Provides a simple interface for emitting events.

This class provides a static method for emitting events, which logs the event to the [EventManager](#).

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

7.9.2 Member Function Documentation

7.9.2.1 `emit()`

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

Emits an event.

Parameters

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

Template Parameters

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

Definition at line [275](#) of file [event_manager.h](#).

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

- `lib/eventman/event_manager.h`

7.10 EventLog Class Reference

Structure for storing event log data.

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

Public Attributes

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

7.10.1 Detailed Description

Structure for storing event log data.

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

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

7.10.2 Member Data Documentation

7.10.2.1 id

```
uint16_t EventLog::id
```

Unique identifier for the event.

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

7.10.2.2 timestamp

```
uint32_t EventLog::timestamp
```

Timestamp of the event in milliseconds since boot.

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

7.10.2.3 group

```
uint8_t EventLog::group
```

Event group.

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

7.10.2.4 event

```
uint8_t EventLog::event
```

Event code.

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

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

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

7.11 EventManager Class Reference

Manages event logging and storage.

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

Public Member Functions

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

Static Public Member Functions

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

Private Member Functions

- [EventManager](#) ()
- [EventManager](#) ([const](#) [EventManager](#) &)=delete
- [EventManager](#) & [operator=](#) ([const](#) [EventManager](#) &)=delete

Private Attributes

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

7.11.1 Detailed Description

Manages event logging and storage.

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

Definition at line 190 of file [event_manager.h](#).

7.11.2 Constructor & Destructor Documentation

7.11.2.1 EventManager() [1/2]

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

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

7.11.2.2 EventManager() [2/2]

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

7.11.3 Member Function Documentation

7.11.3.1 operator=()

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

7.11.3.2 get_instance()

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

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

Returns

A reference to the singleton instance.

Definition at line 216 of file [event_manager.h](#).

7.11.3.3 `get_event_count()`

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

Gets the number of events in the buffer.

Returns

The number of events in the buffer.

Definition at line 245 of file [event_manager.h](#).

7.11.3.4 `load_from_storage()`

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

Loads the event buffer from persistent storage.

Returns

True if the load was successful, false otherwise.

7.11.4 Member Data Documentation

7.11.4.1 `events`

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

Definition at line 192 of file [event_manager.h](#).

7.11.4.2 `eventCount`

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

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

7.11.4.3 `writeIndex`

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

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

7.11.4.4 `eventMutex`

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

Definition at line 195 of file [event_manager.h](#).

7.11.4.5 nextEventId

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

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

7.11.4.6 eventsSinceFlush

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

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

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

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

7.12 Frame Struct Reference

Represents a communication frame used for data exchange.

```
#include <protocol.h>
```

Public Attributes

- `std::string` [header](#)
- `uint8_t` [direction](#)
- `OperationType` [operationType](#)
- `uint8_t` [group](#)
- `uint8_t` [command](#)
- `std::string` [value](#)
- `std::string` [unit](#)
- `std::string` [footer](#)

7.12.1 Detailed Description

Represents a communication frame used for data exchange.

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

Note

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

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

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

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

The `value` field contains the payload data.

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

Definition at line 181 of file [protocol.h](#).

7.12.2 Member Data Documentation

7.12.2.1 header

`std::string Frame::header`

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

7.12.2.2 direction

`uint8_t Frame::direction`

Definition at line 183 of file [protocol.h](#).

7.12.2.3 operationType

`OperationType Frame::operationType`

Definition at line 184 of file [protocol.h](#).

7.12.2.4 group

`uint8_t Frame::group`

Definition at line 185 of file [protocol.h](#).

7.12.2.5 command

`uint8_t Frame::command`

Definition at line 186 of file [protocol.h](#).

7.12.2.6 value

`std::string Frame::value`

Definition at line 187 of file [protocol.h](#).

7.12.2.7 unit

`std::string Frame::unit`

Definition at line 188 of file [protocol.h](#).

7.12.2.8 footer

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

Definition at line 189 of file [protocol.h](#).

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

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

7.13 INA3221 Class Reference

[INA3221](#) Triple-Channel Power Monitor driver class.

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

Classes

- struct [conf_reg_t](#)
Configuration register bit fields.
- struct [masken_reg_t](#)
Mask/Enable register bit fields.

Public Member Functions

- [INA3221](#) ([ina3221_addr_t](#) addr, [i2c_inst_t](#) *i2c)
Constructor for [INA3221](#) class.
- bool [begin](#) ()
Initialize the [INA3221](#) device.
- [uint16_t](#) [read_register](#) ([ina3221_reg_t](#) reg)
Read a register from the device.
- void [reset](#) ()
Reset the [INA3221](#) to default settings.
- void [set_mode_power_down](#) ()
Set device to power-down mode.
- void [set_mode_continuous](#) ()
Set device to continuous measurement mode.
- void [set_mode_triggered](#) ()
Set device to triggered measurement mode.
- void [set_shunt_measurement_enable](#) ()
Enable shunt voltage measurements.
- void [set_shunt_measurement_disable](#) ()
Disable shunt voltage measurements.
- void [set_bus_measurement_enable](#) ()
Enable bus voltage measurements.
- void [set_bus_measurement_disable](#) ()
Disable bus voltage measurements.
- void [set_averaging_mode](#) ([ina3221_avg_mode_t](#) mode)

- Set the averaging mode for measurements.*

 - void [set_bus_conversion_time](#) ([ina3221_conv_time_t](#) convTime)

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

Set shunt voltage conversion time.
- [uint16_t](#) [get_manufacturer_id](#) ()

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

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

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

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

Get bus voltage for a specific channel.

Private Member Functions

- void [_read](#) ([ina3221_reg_t](#) reg, [uint16_t](#) *val)

Read a 16-bit register from the device.
- void [_write](#) ([ina3221_reg_t](#) reg, [uint16_t](#) *val)

Write a 16-bit value to a register.

Private Attributes

- [ina3221_addr_t](#) [_i2c_addr](#)
- [i2c_inst_t](#) * [_i2c](#)
- [uint32_t](#) [_shuntRes](#) [[INA3221_CH_NUM](#)]
- [uint32_t](#) [_filterRes](#) [[INA3221_CH_NUM](#)]
- [masken_reg_t](#) [_masken_reg](#)

7.13.1 Detailed Description

[INA3221](#) Triple-Channel Power Monitor driver class.

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

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

7.13.2 Member Function Documentation

7.13.2.1 [_read\(\)](#)

```
void INA3221::_read (
    ina3221\_reg\_t reg,
    uint16\_t * val) [private]
```

Read a 16-bit register from the device.

Parameters

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

Definition at line 354 of file [INA3221.cpp](#).

7.13.2.2 `_write()`

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

Write a 16-bit value to a register.

Parameters

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

Definition at line 380 of file [INA3221.cpp](#).

7.13.2.3 `get_current()`

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

7.13.3 Member Data Documentation**7.13.3.1 `_i2c_addr`**

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

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

7.13.3.2 `_i2c`

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

Definition at line 138 of file [INA3221.h](#).

7.13.3.3 `_shuntRes`

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

Definition at line 141 of file [INA3221.h](#).

7.13.3.4 `_filterRes`

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

Definition at line 144 of file [INA3221.h](#).

7.13.3.5 `_masken_reg`

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

Definition at line 147 of file [INA3221.h](#).

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

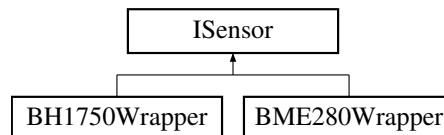
- lib/powerman/INA3221/[INA3221.h](#)
- lib/powerman/INA3221/[INA3221.cpp](#)

7.14 ISensor Class Reference

Abstract base class for sensors.

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

Inheritance diagram for ISensor:



Public Member Functions

- virtual `~ISensor()`=default
Virtual destructor.
- virtual bool `init()`=0
Initializes the sensor.
- virtual float `read_data(SensorDataTypeIdIdentifier type)`=0
Reads data from the sensor.
- virtual bool `is_initialized()` const =0
Checks if the sensor is initialized.
- virtual `SensorType get_type()` const =0
Gets the sensor type.
- virtual bool `configure` (const std::map< std::string, std::string > &config)=0
Configures the sensor.
- virtual uint8_t `get_address()` const =0
Gets the I2C address of the sensor.

7.14.1 Detailed Description

Abstract base class for sensors.

Defines the interface for interacting with different types of sensors.

Definition at line 63 of file [ISensor.h](#).

7.14.2 Constructor & Destructor Documentation

7.14.2.1 ~ISensor()

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

Virtual destructor.

Ensures proper cleanup of derived classes.

7.14.3 Member Function Documentation

7.14.3.1 init()

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

Initializes the sensor.

Returns

True if initialization was successful, false otherwise.

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

7.14.3.2 read_data()

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

Reads data from the sensor.

Parameters

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

Returns

The sensor data.

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

7.14.3.3 is_initialized()

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

Checks if the sensor is initialized.

Returns

True if the sensor is initialized, false otherwise.

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

7.14.3.4 get_type()

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

Gets the sensor type.

Returns

The sensor type.

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

7.14.3.5 configure()

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

Configures the sensor.

Parameters

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

Returns

True if configuration was successful, false otherwise.

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

7.14.3.6 get_address()

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

Gets the I2C address of the sensor.

Returns

The I2C address of the sensor.

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

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

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

7.15 INA3221::masken_reg_t Struct Reference

Mask/Enable register bit fields.

Public Attributes

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

7.15.1 Detailed Description

Mask/Enable register bit fields.

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

7.15.2 Member Data Documentation

7.15.2.1 conv_ready

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

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

7.15.2.2 timing_ctrl_alert

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

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

7.15.2.3 pwr_valid_alert

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

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

7.15.2.4 warn_alert_ch3

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

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

7.15.2.5 warn_alert_ch2

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

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

7.15.2.6 warn_alert_ch1

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

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

7.15.2.7 shunt_sum_alert

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

Definition at line 124 of file [INA3221.h](#).

7.15.2.8 crit_alert_ch3

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

Definition at line 125 of file [INA3221.h](#).

7.15.2.9 crit_alert_ch2

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

Definition at line 126 of file [INA3221.h](#).

7.15.2.10 crit_alert_ch1

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

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

7.15.2.11 crit_alert_latch_en

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

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

7.15.2.12 warn_alert_latch_en

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

Definition at line 129 of file [INA3221.h](#).

7.15.2.13 shunt_sum_en_ch3

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

Definition at line 130 of file [INA3221.h](#).

7.15.2.14 shunt_sum_en_ch2

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

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

7.15.2.15 shunt_sum_en_ch1

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

Definition at line 132 of file [INA3221.h](#).

7.15.2.16 reserved

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

Definition at line 133 of file [INA3221.h](#).

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

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

7.16 NMEADData Class Reference

Manages parsed NMEA sentences.

```
#include <NMEA_data.h>
```

Public Member Functions

- void [update_rmc_tokens](#) (const std::vector< std::string > &tokens)
Updates the RMC tokens with new data.
- void [update_gga_tokens](#) (const std::vector< std::string > &tokens)
Updates the GGA tokens with new data.
- std::vector< std::string > [get_rmc_tokens](#) () const
Gets a copy of the RMC tokens.
- std::vector< std::string > [get_gga_tokens](#) () const
Gets a copy of the GGA tokens.
- bool [has_valid_time](#) () const
Checks if the NMEA data has valid time information.
- time_t [get_unix_time](#) () const
Converts the NMEA data to a Unix timestamp.

Static Public Member Functions

- static [NMEADData](#) & [get_instance](#) ()
Gets the singleton instance of the [NMEADData](#) class.

Private Member Functions

- [NMEADData](#) ()
Private constructor for the singleton pattern.
- [NMEADData](#) (const [NMEADData](#) &)=delete
Deleted copy constructor to prevent copying.
- [NMEADData](#) & [operator=](#) (const [NMEADData](#) &)=delete
Deleted assignment operator to prevent assignment.

Private Attributes

- std::vector< std::string > [rmc_tokens_](#)
Vector of tokens from the most recent RMC sentence.
- std::vector< std::string > [gga_tokens_](#)
Vector of tokens from the most recent GGA sentence.
- mutex_t [rmc_mutex_](#)
Mutex for thread-safe access to the RMC tokens.
- mutex_t [gga_mutex_](#)
Mutex for thread-safe access to the GGA tokens.

7.16.1 Detailed Description

Manages parsed NMEA sentences.

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

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

7.16.2 Constructor & Destructor Documentation

7.16.2.1 NMEADData() [1/2]

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

Private constructor for the singleton pattern.

Initializes the mutexes.

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

7.16.2.2 NMEADData() [2/2]

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

Deleted copy constructor to prevent copying.

7.16.3 Member Function Documentation

7.16.3.1 operator=()

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

Deleted assignment operator to prevent assignment.

7.16.3.2 get_instance()

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

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

Returns

A reference to the singleton instance.

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

7.16.3.3 update_rmc_tokens()

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

Updates the RMC tokens with new data.

Parameters

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

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

7.16.3.4 update_gga_tokens()

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

Updates the GGA tokens with new data.

Parameters

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

Definition at line 86 of file [NMEA_data.h](#).

7.16.3.5 get_rmc_tokens()

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

Gets a copy of the RMC tokens.

Returns

A copy of the RMC tokens.

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

7.16.3.6 get_gga_tokens()

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

Gets a copy of the GGA tokens.

Returns

A copy of the GGA tokens.

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

7.16.3.7 has_valid_time()

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

Checks if the NMEA data has valid time information.

Returns

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

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

7.16.3.8 get_unix_time()

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

Converts the NMEA data to a Unix timestamp.

Returns

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

Definition at line 126 of file [NMEA_data.h](#).

7.16.4 Member Data Documentation

7.16.4.1 rmc_tokens_

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

Vector of tokens from the most recent RMC sentence.

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

7.16.4.2 gga_tokens_

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

Vector of tokens from the most recent GGA sentence.

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

7.16.4.3 rmc_mutex_

```
mutex_t NMEADData::rmc_mutex_ [private]
```

Mutex for thread-safe access to the RMC tokens.

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

7.16.4.4 gga_mutex_

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

Mutex for thread-safe access to the GGA tokens.

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

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

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

7.17 PowerManager Class Reference

Manages power-related functions.

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

Public Member Functions

- [PowerManager](#) (i2c_inst_t *i2c)
Constructor for the [PowerManager](#) class.
- bool [initialize](#) ()
Initializes the [PowerManager](#).
- std::string [read_device_ids](#) ()
Reads the manufacturer and die IDs from the [INA3221](#).
- float [get_current_charge_solar](#) ()
Gets the solar charging current.
- float [get_current_charge_usb](#) ()
Gets the USB charging current.
- float [get_current_charge_total](#) ()
Gets the total charging current.
- float [get_current_draw](#) ()
Gets the current draw.
- float [get_voltage_battery](#) ()
Gets the battery voltage.
- float [get_voltage_5v](#) ()
Gets the 5V voltage.
- void [configure](#) (const std::map< std::string, std::string > &config)
Configures the [INA3221](#).
- bool [is_charging_solar](#) ()
Checks if solar charging is active.
- bool [is_charging_usb](#) ()
Checks if USB charging is active.

Static Public Member Functions

- static [PowerManager](#) & [get_instance](#) ()
Gets the singleton instance of the [PowerManager](#) class.

Static Public Attributes

- static constexpr float [SOLAR_CURRENT_THRESHOLD](#) = 50.0f
Solar current threshold in milliamperes.
- static constexpr float [USB_CURRENT_THRESHOLD](#) = 50.0f
USB current threshold in milliamperes.
- static constexpr float [BATTERY_LOW_THRESHOLD](#) = 2.8f
Low voltage threshold in volts.
- static constexpr float [BATTERY_FULL_THRESHOLD](#) = 4.2f
Overcharge voltage threshold in volts.

Private Member Functions

- [PowerManager](#) ()
Private constructor for the singleton pattern.
- [PowerManager](#) (const [PowerManager](#) &)=delete
Deleted copy constructor to prevent copying.
- [PowerManager](#) & operator= (const [PowerManager](#) &)=delete
Deleted assignment operator to prevent assignment.

Private Attributes

- [INA3221](#) [ina3221_](#)
INA3221 instance for power monitoring.
- bool [initialized_](#)
Flag indicating if the [PowerManager](#) is initialized.
- recursive_mutex_t [powerman_mutex_](#)
Mutex for thread-safe access to the [PowerManager](#).
- bool [charging_solar_active_](#) = false
Flag indicating if solar charging is active.
- bool [charging_usb_active_](#) = false
Flag indicating if USB charging is active.

7.17.1 Detailed Description

Manages power-related functions.

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

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

7.17.2 Constructor & Destructor Documentation

7.17.2.1 [PowerManager](#)() [1/2]

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

Constructor for the [PowerManager](#) class.

Parameters

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

7.17.2.2 `PowerManager()` [2/2]

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

Deleted copy constructor to prevent copying.

7.17.3 Member Function Documentation

7.17.3.1 `operator=()`

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

Deleted assignment operator to prevent assignment.

7.17.4 Member Data Documentation

7.17.4.1 `SOLAR_CURRENT_THRESHOLD`

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

Solar current threshold in milliamperes.

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

7.17.4.2 `USB_CURRENT_THRESHOLD`

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

USB current threshold in milliamperes.

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

7.17.4.3 `BATTERY_LOW_THRESHOLD`

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

Low voltage threshold in volts.

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

7.17.4.4 BATTERY_FULL_THRESHOLD

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

Overcharge voltage threshold in volts.

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

7.17.4.5 ina3221_

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

[INA3221](#) instance for power monitoring.

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

7.17.4.6 initialized_

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

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

Definition at line 126 of file [PowerManager.h](#).

7.17.4.7 powerman_mutex_

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

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

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

7.17.4.8 charging_solar_active_

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

Flag indicating if solar charging is active.

Definition at line 130 of file [PowerManager.h](#).

7.17.4.9 charging_usb_active_

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

Flag indicating if USB charging is active.

Definition at line 132 of file [PowerManager.h](#).

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

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

7.18 SensorDataRecord Struct Reference

Structure representing a single sensor data point.

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

Public Member Functions

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

Public Attributes

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

7.18.1 Detailed Description

Structure representing a single sensor data point.

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

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

7.18.2 Member Data Documentation

7.18.2.1 timestamp

```
uint32_t SensorDataRecord::timestamp
```

Unix timestamp of the record

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

7.18.2.2 temperature

```
float SensorDataRecord::temperature
```

Temperature in degrees Celsius

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

7.18.2.3 pressure

```
float SensorDataRecord::pressure
```

Pressure in hPa

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

7.18.2.4 humidity

```
float SensorDataRecord::humidity
```

Relative humidity in %

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

7.18.2.5 light

```
float SensorDataRecord::light
```

Light intensity in lux

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

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

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

7.19 SensorWrapper Class Reference

Manages a collection of sensors.

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

Public Member Functions

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

Static Public Member Functions

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

Private Member Functions

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

Private Attributes

- std::map< [SensorType](#), [ISensor](#) * > [sensors](#)
Map of sensor types to sensor instances.

7.19.1 Detailed Description

Manages a collection of sensors.

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

Definition at line 116 of file [ISensor.h](#).

7.19.2 Constructor & Destructor Documentation

7.19.2.1 SensorWrapper()

```
SensorWrapper::SensorWrapper () [private], [default]
```

Private constructor for the singleton pattern.

7.19.3 Member Function Documentation

7.19.3.1 get_instance()

```
static SensorWrapper & SensorWrapper::get_instance () [inline], [static]
```

Gets the singleton instance of the [SensorWrapper](#) class.

Returns

A reference to the singleton instance.

Definition at line 122 of file [ISensor.h](#).

7.19.4 Member Data Documentation

7.19.4.1 sensors

```
std::map<SensorType, ISensor*> SensorWrapper::sensors [private]
```

Map of sensor types to sensor instances.

Definition at line 173 of file [ISensor.h](#).

The documentation for this class was generated from the following files:

- [lib/sensors/ISensor.h](#)
- [lib/sensors/ISensor.cpp](#)

7.20 SystemStateManager Class Reference

Manages the system state of the Kubisat firmware.

```
#include <system_state_manager.h>
```

Public Member Functions

- bool [is_bootloader_reset_pending](#) () const
Checks if a bootloader reset is pending.
- void [set_bootloader_reset_pending](#) (bool pending)
Sets whether a bootloader reset is pending.
- bool [is_gps_collection_paused](#) () const
Checks if GPS collection is paused.
- void [set_gps_collection_paused](#) (bool paused)
Sets whether GPS collection is paused.
- bool [is_sd_card_mounted](#) () const
Checks if the SD card is mounted.
- void [set_sd_card_mounted](#) (bool mounted)
Sets whether the SD card is mounted.
- [VerbosityLevel](#) [get_uart_verbosity](#) () const
Gets the UART verbosity level.
- void [set_uart_verbosity](#) ([VerbosityLevel](#) level)
Sets the UART verbosity level.
- bool [is_radio_init_ok](#) () const
Checks if the radio initialization was successful.
- void [set_radio_init_ok](#) (bool status)
Sets whether the radio initialization was successful.
- bool [is_light_sensor_init_ok](#) () const
Checks if the light sensor initialization was successful.
- void [set_light_sensor_init_ok](#) (bool status)
Sets whether the light sensor initialization was successful.
- bool [is_env_sensor_init_ok](#) () const
Checks if the environment sensor initialization was successful.
- void [set_env_sensor_init_ok](#) (bool status)
Sets whether the environment sensor initialization was successful.

Static Public Member Functions

- static [SystemStateManager](#) & [get_instance](#) ()
Gets the singleton instance of the [SystemStateManager](#) class.

Private Member Functions

- [SystemStateManager](#) ()
Private constructor for the singleton pattern.
- [SystemStateManager](#) (const [SystemStateManager](#) &)=delete
Deleted copy constructor to prevent copying.
- [SystemStateManager](#) & [operator=](#) (const [SystemStateManager](#) &)=delete
Deleted assignment operator to prevent assignment.

Private Attributes

- bool [pending_bootloader_reset](#)
Flag indicating whether a bootloader reset is pending.
- bool [gps_collection_paused](#)
Flag indicating whether GPS collection is paused.
- bool [sd_card_mounted](#)
Flag indicating whether the SD card is mounted.
- [VerbosityLevel](#) [uart_verbosity](#)
The UART verbosity level.
- bool [sd_card_init_status](#)
Flag indicating whether the SD card initialization was successful.
- bool [radio_init_status](#)
Flag indicating whether the radio initialization was successful.
- bool [light_sensor_init_status](#)
Flag indicating whether the light sensor initialization was successful.
- bool [env_sensor_init_status](#)
Flag indicating whether the environment sensor initialization was successful.
- recursive_mutex_t [mutex](#)
Mutex for thread-safe access to the system state.

7.20.1 Detailed Description

Manages the system state of the Kubisat firmware.

This class is a singleton that provides methods for getting and setting various system states, such as whether a bootloader reset is pending, whether GPS collection is paused, whether the SD card is mounted, and the UART verbosity level.

Definition at line 32 of file [system_state_manager.h](#).

7.20.2 Constructor & Destructor Documentation

7.20.2.1 SystemStateManager() [1/2]

```
SystemStateManager::SystemStateManager () [inline], [private]
```

Private constructor for the singleton pattern.

Initializes the system state and mutex.

Definition at line 57 of file [system_state_manager.h](#).

7.20.2.2 SystemStateManager() [2/2]

```
SystemStateManager::SystemStateManager (
    const SystemStateManager & ) [private], [delete]
```

Deleted copy constructor to prevent copying.

7.20.3 Member Function Documentation

7.20.3.1 operator=()

```
SystemStateManager & SystemStateManager::operator= (
    const SystemStateManager & ) [private], [delete]
```

Deleted assignment operator to prevent assignment.

7.20.3.2 get_instance()

```
static SystemStateManager & SystemStateManager::get_instance () [inline], [static]
```

Gets the singleton instance of the [SystemStateManager](#) class.

Returns

A reference to the singleton instance.

Definition at line 84 of file [system_state_manager.h](#).

7.20.3.3 is_bootloader_reset_pending()

```
bool SystemStateManager::is_bootloader_reset_pending () const [inline]
```

Checks if a bootloader reset is pending.

Returns

True if a bootloader reset is pending, false otherwise.

Definition at line 93 of file [system_state_manager.h](#).

7.20.3.4 set_bootloader_reset_pending()

```
void SystemStateManager::set_bootloader_reset_pending (
    bool pending) [inline]
```

Sets whether a bootloader reset is pending.

Parameters

in	<i>pending</i>	True if a bootloader reset is pending, false otherwise.
----	----------------	---

Definition at line 104 of file [system_state_manager.h](#).

7.20.3.5 is_gps_collection_paused()

```
bool SystemStateManager::is_gps_collection_paused () const [inline]
```

Checks if GPS collection is paused.

Returns

True if GPS collection is paused, false otherwise.

Definition at line 114 of file [system_state_manager.h](#).

7.20.3.6 set_gps_collection_paused()

```
void SystemStateManager::set_gps_collection_paused (  
    bool paused) [inline]
```

Sets whether GPS collection is paused.

Parameters

in	<i>paused</i>	True if GPS collection is paused, false otherwise.
----	---------------	--

Definition at line 125 of file [system_state_manager.h](#).

7.20.3.7 is_sd_card_mounted()

```
bool SystemStateManager::is_sd_card_mounted () const [inline]
```

Checks if the SD card is mounted.

Returns

True if the SD card is mounted, false otherwise.

Definition at line 135 of file [system_state_manager.h](#).

7.20.3.8 set_sd_card_mounted()

```
void SystemStateManager::set_sd_card_mounted (  
    bool mounted) [inline]
```

Sets whether the SD card is mounted.

Parameters

<i>in</i>	<i>mounted</i>	True if the SD card is mounted, false otherwise.
-----------	----------------	--

Definition at line 146 of file [system_state_manager.h](#).

7.20.3.9 get_uart_verbosity()

```
VerbosityLevel SystemStateManager::get_uart_verbosity () const [inline]
```

Gets the UART verbosity level.

Returns

The UART verbosity level.

Definition at line 156 of file [system_state_manager.h](#).

7.20.3.10 set_uart_verbosity()

```
void SystemStateManager::set_uart_verbosity (  
    VerbosityLevel level) [inline]
```

Sets the UART verbosity level.

Parameters

<i>in</i>	<i>level</i>	The UART verbosity level.
-----------	--------------	---------------------------

Definition at line 167 of file [system_state_manager.h](#).

7.20.3.11 is_radio_init_ok()

```
bool SystemStateManager::is_radio_init_ok () const [inline]
```

Checks if the radio initialization was successful.

Returns

True if the radio initialization was successful, false otherwise.

Definition at line 177 of file [system_state_manager.h](#).

7.20.3.12 set_radio_init_ok()

```
void SystemStateManager::set_radio_init_ok (  
    bool status) [inline]
```

Sets whether the radio initialization was successful.

Parameters

<i>in</i>	<i>status</i>	True if the radio initialization was successful, false otherwise.
-----------	---------------	---

Definition at line 188 of file [system_state_manager.h](#).

7.20.3.13 is_light_sensor_init_ok()

```
bool SystemStateManager::is_light_sensor_init_ok () const [inline]
```

Checks if the light sensor initialization was successful.

Returns

True if the light sensor initialization was successful, false otherwise.

Definition at line 198 of file [system_state_manager.h](#).

7.20.3.14 set_light_sensor_init_ok()

```
void SystemStateManager::set_light_sensor_init_ok (  
    bool status) [inline]
```

Sets whether the light sensor initialization was successful.

Parameters

<i>in</i>	<i>status</i>	True if the light sensor initialization was successful, false otherwise.
-----------	---------------	--

Definition at line 209 of file [system_state_manager.h](#).

7.20.3.15 is_env_sensor_init_ok()

```
bool SystemStateManager::is_env_sensor_init_ok () const [inline]
```

Checks if the environment sensor initialization was successful.

Returns

True if the environment sensor initialization was successful, false otherwise.

Definition at line 219 of file [system_state_manager.h](#).

7.20.3.16 set_env_sensor_init_ok()

```
void SystemStateManager::set_env_sensor_init_ok (  
    bool status) [inline]
```

Sets whether the environment sensor initialization was successful.

Parameters

<code>in</code>	<code>status</code>	True if the environment sensor initialization was successful, false otherwise.
-----------------	---------------------	--

Definition at line 230 of file [system_state_manager.h](#).

7.20.4 Member Data Documentation

7.20.4.1 pending_bootloader_reset

```
bool SystemStateManager::pending_bootloader_reset [private]
```

Flag indicating whether a bootloader reset is pending.

Definition at line 35 of file [system_state_manager.h](#).

7.20.4.2 gps_collection_paused

```
bool SystemStateManager::gps_collection_paused [private]
```

Flag indicating whether GPS collection is paused.

Definition at line 37 of file [system_state_manager.h](#).

7.20.4.3 sd_card_mounted

```
bool SystemStateManager::sd_card_mounted [private]
```

Flag indicating whether the SD card is mounted.

Definition at line 39 of file [system_state_manager.h](#).

7.20.4.4 uart_verbosity

```
VerbosityLevel SystemStateManager::uart_verbosity [private]
```

The UART verbosity level.

Definition at line 41 of file [system_state_manager.h](#).

7.20.4.5 sd_card_init_status

```
bool SystemStateManager::sd_card_init_status [private]
```

Flag indicating whether the SD card initialization was successful.

Definition at line 43 of file [system_state_manager.h](#).

7.20.4.6 radio_init_status

```
bool SystemStateManager::radio_init_status [private]
```

Flag indicating whether the radio initialization was successful.

Definition at line 45 of file [system_state_manager.h](#).

7.20.4.7 light_sensor_init_status

```
bool SystemStateManager::light_sensor_init_status [private]
```

Flag indicating whether the light sensor initialization was successful.

Definition at line 47 of file [system_state_manager.h](#).

7.20.4.8 env_sensor_init_status

```
bool SystemStateManager::env_sensor_init_status [private]
```

Flag indicating whether the environment sensor initialization was successful.

Definition at line 49 of file [system_state_manager.h](#).

7.20.4.9 mutex_

```
recursive_mutex_t SystemStateManager::mutex_ [private]
```

Mutex for thread-safe access to the system state.

Definition at line 51 of file [system_state_manager.h](#).

The documentation for this class was generated from the following file:

- [lib/system_state_manager.h](#)

7.21 TelemetryManager Class Reference

Manages the collection, storage, and retrieval of telemetry data.

```
#include <telemetry_manager.h>
```

Public Member Functions

- `bool init ()`
Initialize the telemetry system.
- `bool collect_telemetry ()`
Collect telemetry data from sensors and power subsystems.
- `void collect_power_telemetry (TelemetryRecord &record)`
Collects power subsystem telemetry data.
- `void emit_power_events (float battery_voltage, float charge_current_usb, float charge_current_solar)`
Emits power-related events based on current and voltage levels.
- `void collect_gps_telemetry (TelemetryRecord &record)`
Collects GPS telemetry data.
- `void collect_sensor_telemetry (SensorDataRecord &sensor_record)`
Collects sensor telemetry data.
- `bool flush_telemetry ()`
Save buffered telemetry data to storage.
- `bool flush_sensor_data ()`
Save buffered sensor data to storage.
- `bool is_telemetry_collection_time (uint32_t current_time, uint32_t &last_collection_time)`
Check if it's time to collect telemetry based on interval.
- `bool is_telemetry_flush_time (uint32_t &collection_counter)`
Check if it's time to flush telemetry buffer based on count.
- `std::string get_last_telemetry_record_csv ()`
Gets the last telemetry record as a CSV string.
- `std::string get_last_sensor_record_csv ()`
Gets the last sensor data record as a CSV string.
- `TelemetryRecord & get_last_telemetry_record ()`
- `SensorDataRecord & get_last_sensor_record ()`
- `size_t get_telemetry_buffer_count () const`
- `size_t get_telemetry_buffer_write_index () const`

Static Public Member Functions

- `static TelemetryManager & get_instance ()`
Gets the singleton instance of the [TelemetryManager](#) class.

Static Public Attributes

- `static constexpr int TELEMETRY_BUFFER_SIZE = 20`

Private Member Functions

- `TelemetryManager ()`
- `~TelemetryManager ()=default`

Private Attributes

- uint32_t [sample_interval_ms](#) = [DEFAULT_SAMPLE_INTERVAL_MS](#)
- uint32_t [flush_threshold](#) = [DEFAULT_FLUSH_THRESHOLD](#)
Current flush threshold (number of records that triggers a flush)
- [TelemetryRecord](#) [telemetry_buffer](#) [[TELEMETRY_BUFFER_SIZE](#)]
Circular buffer for telemetry records.
- size_t [telemetry_buffer_count](#) = 0
- size_t [telemetry_buffer_write_index](#) = 0
- [SensorDataRecord](#) [sensor_data_buffer](#) [[TELEMETRY_BUFFER_SIZE](#)]
Circular buffer for sensor data records.
- mutex_t [telemetry_mutex](#)
Mutex for thread-safe access to the telemetry buffer.

Static Private Attributes

- static constexpr uint32_t [DEFAULT_SAMPLE_INTERVAL_MS](#) = 1000
Current sampling interval in milliseconds.
- static constexpr uint32_t [DEFAULT_FLUSH_THRESHOLD](#) = 10
Default number of records to collect before flushing to storage.

7.21.1 Detailed Description

Manages the collection, storage, and retrieval of telemetry data.

This class implements a singleton pattern to provide a single point of access for managing telemetry data. It handles the collection of data from various subsystems, stores the data in circular buffers, and provides methods for flushing the data to persistent storage and retrieving the last recorded data.

Definition at line [145](#) of file [telemetry_manager.h](#).

7.21.2 Constructor & Destructor Documentation

7.21.2.1 ~TelemetryManager()

```
TelemetryManager::~TelemetryManager () [private], [default]
```

7.21.3 Member Function Documentation

7.21.3.1 get_instance()

```
static TelemetryManager & TelemetryManager::get_instance () [inline], [static]
```

Gets the singleton instance of the [TelemetryManager](#) class.

Returns

A reference to the singleton instance.

Definition at line [151](#) of file [telemetry_manager.h](#).

7.21.3.2 flush_sensor_data()

```
bool TelemetryManager::flush_sensor_data ()
```

Save buffered sensor data to storage.

Returns

True if data was successfully saved

Writes all records from the sensor data buffer to the CSV file and clears the buffer after successful writing

7.21.3.3 get_last_telemetry_record()

```
TelemetryRecord & TelemetryManager::get_last_telemetry_record () [inline]
```

Definition at line 250 of file [telemetry_manager.h](#).

7.21.3.4 get_last_sensor_record()

```
SensorDataRecord & TelemetryManager::get_last_sensor_record () [inline]
```

Definition at line 255 of file [telemetry_manager.h](#).

7.21.3.5 get_telemetry_buffer_count()

```
size_t TelemetryManager::get_telemetry_buffer_count () const [inline]
```

Definition at line 260 of file [telemetry_manager.h](#).

7.21.3.6 get_telemetry_buffer_write_index()

```
size_t TelemetryManager::get_telemetry_buffer_write_index () const [inline]
```

Definition at line 261 of file [telemetry_manager.h](#).

7.21.4 Member Data Documentation

7.21.4.1 TELEMETRY_BUFFER_SIZE

```
int TelemetryManager::TELEMETRY_BUFFER_SIZE = 20 [static], [constexpr]
```

Definition at line 248 of file [telemetry_manager.h](#).

7.21.4.2 DEFAULT_SAMPLE_INTERVAL_MS

```
uint32_t TelemetryManager::DEFAULT_SAMPLE_INTERVAL_MS = 1000 [static], [constexpr], [private]
```

Current sampling interval in milliseconds.

Definition at line 270 of file [telemetry_manager.h](#).

7.21.4.3 DEFAULT_FLUSH_THRESHOLD

```
uint32_t TelemetryManager::DEFAULT_FLUSH_THRESHOLD = 10 [static], [constexpr], [private]
```

Default number of records to collect before flushing to storage.

Definition at line 275 of file [telemetry_manager.h](#).

7.21.4.4 sample_interval_ms

```
uint32_t TelemetryManager::sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS [private]
```

Definition at line 277 of file [telemetry_manager.h](#).

7.21.4.5 flush_threshold

```
uint32_t TelemetryManager::flush_threshold = DEFAULT_FLUSH_THRESHOLD [private]
```

Current flush threshold (number of records that triggers a flush)

Definition at line 282 of file [telemetry_manager.h](#).

7.21.4.6 telemetry_buffer

```
TelemetryRecord TelemetryManager::telemetry_buffer[TELEMETRY_BUFFER_SIZE] [private]
```

Circular buffer for telemetry records.

Definition at line 287 of file [telemetry_manager.h](#).

7.21.4.7 telemetry_buffer_count

```
size_t TelemetryManager::telemetry_buffer_count = 0 [private]
```

Definition at line 288 of file [telemetry_manager.h](#).

7.21.4.8 telemetry_buffer_write_index

```
size_t TelemetryManager::telemetry_buffer_write_index = 0 [private]
```

Definition at line 289 of file [telemetry_manager.h](#).

7.21.4.9 sensor_data_buffer

`SensorDataRecord` `TelemetryManager::sensor_data_buffer[TELEMETRY_BUFFER_SIZE]` [private]

Circular buffer for sensor data records.

Definition at line 294 of file [telemetry_manager.h](#).

7.21.4.10 telemetry_mutex

`mutex_t` `TelemetryManager::telemetry_mutex` [private]

Mutex for thread-safe access to the telemetry buffer.

Definition at line 299 of file [telemetry_manager.h](#).

The documentation for this class was generated from the following files:

- [lib/telemetry/telemetry_manager.h](#)
- [lib/telemetry/telemetry_manager.cpp](#)

7.22 TelemetryRecord Struct Reference

Structure representing a single telemetry data point.

```
#include <telemetry_manager.h>
```

Public Member Functions

- `std::string` [to_csv](#) () const
Converts the telemetry record to a CSV string.

Public Attributes

- `uint32_t` [timestamp](#)
- `std::string` [build_version](#)
- `float` [battery_voltage](#)
- `float` [system_voltage](#)
- `float` [charge_current_usb](#)
- `float` [charge_current_solar](#)
- `float` [discharge_current](#)
- `std::string` [time](#)
- `std::string` [latitude](#)
- `std::string` [lat_dir](#)
- `std::string` [longitude](#)
- `std::string` [lon_dir](#)
- `std::string` [speed](#)
- `std::string` [course](#)
- `std::string` [date](#)
- `std::string` [fix_quality](#)
- `std::string` [satellites](#)
- `std::string` [altitude](#)

7.22.1 Detailed Description

Structure representing a single telemetry data point.

Contains all measurements from power subsystem, sensors, and GPS data collected at a specific point in time

Definition at line 43 of file [telemetry_manager.h](#).

7.22.2 Member Data Documentation

7.22.2.1 timestamp

```
uint32_t TelemetryRecord::timestamp
```

Unix timestamp of the record

Definition at line 44 of file [telemetry_manager.h](#).

7.22.2.2 build_version

```
std::string TelemetryRecord::build_version
```

Build version of the firmware

Definition at line 46 of file [telemetry_manager.h](#).

7.22.2.3 battery_voltage

```
float TelemetryRecord::battery_voltage
```

Battery voltage in volts

Definition at line 49 of file [telemetry_manager.h](#).

7.22.2.4 system_voltage

```
float TelemetryRecord::system_voltage
```

System 5V rail voltage in volts

Definition at line 50 of file [telemetry_manager.h](#).

7.22.2.5 charge_current_usb

```
float TelemetryRecord::charge_current_usb
```

USB charging current in mA

Definition at line 51 of file [telemetry_manager.h](#).

7.22.2.6 charge_current_solar

```
float TelemetryRecord::charge_current_solar
```

Solar charging current in mA

Definition at line 52 of file [telemetry_manager.h](#).

7.22.2.7 discharge_current

```
float TelemetryRecord::discharge_current
```

Battery discharge current in mA

Definition at line 53 of file [telemetry_manager.h](#).

7.22.2.8 time

```
std::string TelemetryRecord::time
```

UTC time from GPS

Definition at line 56 of file [telemetry_manager.h](#).

7.22.2.9 latitude

```
std::string TelemetryRecord::latitude
```

Latitude from GPS

Definition at line 57 of file [telemetry_manager.h](#).

7.22.2.10 lat_dir

```
std::string TelemetryRecord::lat_dir
```

N/S latitude direction

Definition at line 58 of file [telemetry_manager.h](#).

7.22.2.11 longitude

```
std::string TelemetryRecord::longitude
```

Longitude from GPS

Definition at line 59 of file [telemetry_manager.h](#).

7.22.2.12 lon_dir

```
std::string TelemetryRecord::lon_dir
```

E/W longitude direction

Definition at line 60 of file [telemetry_manager.h](#).

7.22.2.13 speed

```
std::string TelemetryRecord::speed
```

Speed in knots

Definition at line 61 of file [telemetry_manager.h](#).

7.22.2.14 course

```
std::string TelemetryRecord::course
```

Course in degrees

Definition at line 62 of file [telemetry_manager.h](#).

7.22.2.15 date

```
std::string TelemetryRecord::date
```

Date from GPS

Definition at line 63 of file [telemetry_manager.h](#).

7.22.2.16 fix_quality

```
std::string TelemetryRecord::fix_quality
```

GPS fix quality

Definition at line 66 of file [telemetry_manager.h](#).

7.22.2.17 satellites

```
std::string TelemetryRecord::satellites
```

Number of satellites in view

Definition at line 67 of file [telemetry_manager.h](#).

7.22.2.18 altitude

```
std::string TelemetryRecord::altitude
```

Altitude in meters

Definition at line 68 of file [telemetry_manager.h](#).

The documentation for this struct was generated from the following file:

- [lib/telemetry/telemetry_manager.h](#)

Chapter 8

File Documentation

8.1 build_number.h File Reference

Macros

- `#define BUILD_NUMBER 496`

8.1.1 Macro Definition Documentation

8.1.1.1 BUILD_NUMBER

```
#define BUILD_NUMBER 496
```

Definition at line 6 of file [build_number.h](#).

8.2 build_number.h

[Go to the documentation of this file.](#)

```
00001 //This file is automatically generated by build_number.cmake
00002
00003 #ifndef CMAKE_BUILD_NUMBER_HEADER
00004 #define CMAKE_BUILD_NUMBER_HEADER
00005
00006 #define BUILD_NUMBER 496
00007
00008 #endif
```

8.3 includes.h File Reference

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "hardware/i2c.h"
#include "hardware/uart.h"
#include "pico/multicore.h"
#include "event_manager.h"
#include "lib/powerman/PowerManager.h"
#include <pico/bootrom.h>
#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/clock/DS3231.h"
#include <iostream>
#include <iomanip>
#include <queue>
#include <chrono>
#include "protocol.h"
#include <atomic>
#include <map>
#include "pin_config.h"
#include "utils.h"
#include "communication.h"
#include "build_number.h"
#include "lib/location/gps_collector.h"
#include "lib/storage/storage.h"
#include "lib/storage/pico-vfs/include/filesystem/vfs.h"
#include "telemetry_manager.h"
#include "system_state_manager.h"
```

8.4 includes.h

[Go to the documentation of this file.](#)

```
00001 #ifndef INCLUDES_H
00002 #define INCLUDES_H
00003
00004 #include <stdio.h>
00005 #include "pico/stdlib.h"
00006 #include "hardware/spi.h"
00007 #include "hardware/i2c.h"
00008 #include "hardware/uart.h"
00009 #include "pico/multicore.h"
00010 #include "event_manager.h"
00011 #include "lib/powerman/PowerManager.h"
00012 #include <pico/bootrom.h>
00013
00014 #include "ISensor.h"
00015 #include "lib/sensors/BH1750/BH1750_WRAPPER.h"
00016 #include "lib/sensors/BME280/BME280_WRAPPER.h"
00017 #include "lib/clock/DS3231.h"
00018 #include <iostream>
00019 #include <iomanip>
00020 #include <queue>
00021 #include <chrono>
00022 #include "protocol.h"
00023 #include <atomic>
00024 #include <iostream>
00025 #include <map>
00026 #include "pin_config.h"
00027 #include "utils.h"
00028 #include "communication.h"
```



```

00029 #include "build_number.h"
00030 #include "lib/location/gps_collector.h"
00031 #include "lib/storage/storage.h"
00032 #include "lib/storage/pico-vfs/include/filesystem/vfs.h"
00033 #include "telemetry_manager.h"
00034 #include "system_state_manager.h"
00035
00036 #endif

```

8.5 lib/clock/DS3231.cpp File Reference

```

#include "DS3231.h"
#include "utils.h"
#include <cstdio>
#include <mutex>
#include "event_manager.h"
#include "NMEA_data.h"

```

8.6 DS3231.cpp

[Go to the documentation of this file.](#)

```

00001 #include "DS3231.h"
00002 #include "utils.h"
00003 #include <cstdio>
00004 #include <mutex>
00005 #include "event_manager.h"
00006 #include "NMEA_data.h"
00007
00013
00023 DS3231::DS3231() : i2c(MAIN_I2C_PORT), ds3231_addr(DS3231_DEVICE_ADDRESS) {
00024     recursive_mutex_init(&clock_mutex_);
00025 }
00026
00027
00038 DS3231& DS3231::get_instance() {
00039     static DS3231 instance;
00040     return instance;
00041 }
00042
00056 int DS3231::set_time(ds3231_data_t *data) {
00057     uint8_t temp[7] = {0};
00058
00059     if (clock_enable() != 0) {
00060         uart_print("Failed to enable clock oscillator", VerbosityLevel::ERROR);
00061         return -1;
00062     }
00063
00064     if (data->seconds > 59)
00065         data->seconds = 59;
00066     if (data->minutes > 59)
00067         data->minutes = 59;
00068     if (data->hours > 23)
00069         data->hours = 23;
00070     if (data->day > 7)
00071         data->day = 7;
00072     else if (data->day < 1)
00073         data->day = 1;
00074     if (data->date > 31)
00075         data->date = 31;
00076     else if (data->date < 1)
00077         data->date = 1;
00078     if (data->month > 12)
00079         data->month = 12;
00080     else if (data->month < 1)
00081         data->month = 1;
00082     if (data->year > 99)
00083         data->year = 99;
00084
00085     temp[0] = bin_to_bcd(data->seconds);
00086     temp[1] = bin_to_bcd(data->minutes);
00087     temp[2] = bin_to_bcd(data->hours);

```

```

00088     temp[2] &= ~(0x01 << 6); // Clear 12/24 hour bit
00089     temp[3] = bin_to_bcd(data->day);
00090     temp[4] = bin_to_bcd(data->date);
00091     temp[5] = bin_to_bcd(data->month);
00092     if (data->century)
00093         temp[5] |= (0x01 << 7);
00094     temp[6] = bin_to_bcd(data->year);
00095
00096     std::string status = "BCD values to be written to DS3231: " + std::to_string(temp[0]) + " " +
00097                         std::to_string(temp[1]) + " " + std::to_string(temp[2]) + " " +
00098                         std::to_string(temp[3]) + " " + std::to_string(temp[4]) + " " +
00099                         std::to_string(temp[5]) + " " + std::to_string(temp[6]);
00100
00101     uart_print(status, VerbosityLevel::DEBUG);
00102
00103     int result = i2c_write_reg(DS3231_SECONDS_REG, 7, temp);
00104     if (result != 0) {
00105         uart_print("i2c write failed", VerbosityLevel::ERROR);
00106         return -1;
00107     }
00108
00109     return 0;
00110 }
00111
00112
00126 int DS3231::get_time(ds3231_data_t *data) {
00127     std::string status;
00128     uint8_t raw_data[7];
00129     int result = i2c_read_reg(DS3231_SECONDS_REG, 7, raw_data);
00130     if (result != 0) {
00131         status = "Failed to read time from DS3231";
00132         uart_print(status, VerbosityLevel::ERROR);
00133         return -1;
00134     }
00135
00136     data->seconds = bcd_to_bin(raw_data[0] & 0x7F); // Masking for CH bit (clock halt)
00137     data->minutes = bcd_to_bin(raw_data[1] & 0x7F);
00138     data->hours = bcd_to_bin(raw_data[2] & 0x3F); // Masking for 12/24 hour mode bit
00139     data->day = raw_data[3] & 0x07; // Day of week (1-7)
00140     data->date = bcd_to_bin(raw_data[4] & 0x3F);
00141     data->month = bcd_to_bin(raw_data[5] & 0x1F); // Masking for century bit
00142     data->century = (raw_data[5] & 0x80) >> 7;
00143     data->year = bcd_to_bin(raw_data[6]);
00144
00145     if (data->seconds > 59 || data->minutes > 59 || data->hours > 23 ||
00146         data->day < 1 || data->day > 7 || data->date < 1 || data->date > 31 ||
00147         data->month < 1 || data->month > 12 || data->year > 99) {
00148         uart_print("Invalid data read from DS3231", VerbosityLevel::ERROR);
00149         return -1;
00150     }
00151
00152     return 0;
00153 }
00154
00155
00168 int DS3231::read_temperature(float *resolution) {
00169     std::string status;
00170     uint8_t temp[2];
00171     int result = i2c_read_reg(DS3231_TEMPERATURE_MSB_REG, 2, temp);
00172     if (result != 0) {
00173         status = "Failed to read temperature from DS3231";
00174         uart_print(status, VerbosityLevel::ERROR);
00175         return -1;
00176     }
00177
00178     int8_t temperature_msb = (int8_t)temp[0];
00179     uint8_t temperature_lsb = temp[1] >> 6; // Only the 2 MSB are valid
00180
00181     *resolution = temperature_msb + (temperature_lsb * 0.25f); // 0.25 degree resolution
00182
00183     return 0;
00184 }
00185
00186
00198 int DS3231::set_unix_time(time_t unix_time) {
00199     struct tm *timeinfo = gmtime(&unix_time);
00200     if (timeinfo == NULL) {
00201         uart_print("Error: gmtime() failed", VerbosityLevel::ERROR);
00202         return -1;
00203     }
00204
00205     ds3231_data_t data;
00206     data.seconds = timeinfo->tm_sec;
00207     data.minutes = timeinfo->tm_min;
00208     data.hours = timeinfo->tm_hour;
00209     data.day = timeinfo->tm_wday == 0 ? 7 : timeinfo->tm_wday; // Sunday is 0 in tm struct, but 1 in
DS3231

```

```

00210     data.date = timeinfo->tm_mday;
00211     data.month = timeinfo->tm_mon + 1; // Month is 0-11 in tm struct, but 1-12 in DS3231
00212     data.year = timeinfo->tm_year - 100; // Year is since 1900, we want the last two digits
00213     data.century = timeinfo->tm_year >= 2000;
00214
00215     return set_time(&data);
00216 }
00217
00218
00229 time_t DS3231::get_unix_time() {
00230     ds3231_data_t data;
00231     if (get_time(&data)) {
00232         return -1;
00233     }
00234
00235     struct tm timeinfo;
00236     timeinfo.tm_sec = data.seconds;
00237     timeinfo.tm_min = data.minutes;
00238     timeinfo.tm_hour = data.hours;
00239     timeinfo.tm_mday = data.date;
00240     timeinfo.tm_mon = data.month - 1; // Month is 0-11 in tm struct, but 1-12 in DS3231
00241     timeinfo.tm_year = data.year + 100; // Year is since 1900
00242
00243     // mktime assumes that tm_wday and tm_yday are uninitialized
00244     timeinfo.tm_wday = 0;
00245     timeinfo.tm_yday = 0;
00246     timeinfo.tm_isdst = 0; // Set to 0 to use UTC
00247
00248     time_t timestamp = mktime(&timeinfo);
00249     if (timestamp == (time_t)(-1)) {
00250         uart_print("Error: mktime() failed", VerbosityLevel::ERROR);
00251         return -1;
00252     }
00253
00254     return timestamp;
00255 }
00256
00257
00267 int DS3231::clock_enable() {
00268     std::string status;
00269     uint8_t control_reg = 0;
00270     int result = i2c_read_reg(DS3231_CONTROL_REG, 1, &control_reg);
00271     if (result != 0) {
00272         status = "Failed to read control register";
00273         uart_print(status, VerbosityLevel::ERROR);
00274         return -1;
00275     }
00276
00277     // Clear the EOSC bit to enable the oscillator
00278     control_reg &= ~(1 << 7);
00279
00280     result = i2c_write_reg(DS3231_CONTROL_REG, 1, &control_reg);
00281     if (result != 0) {
00282         status = "Failed to write control register";
00283         uart_print(status, VerbosityLevel::ERROR);
00284         return -1;
00285     }
00286
00287     return 0;
00288 }
00289
00290
00300 int16_t DS3231::get_timezone_offset() const {
00301     return timezone_offset_minutes_;
00302 }
00303
00304
00316 void DS3231::set_timezone_offset(int16_t offset_minutes) {
00317     // Validate range: -12 hours to +12 hours (-720 to +720 minutes)
00318     if (offset_minutes >= -720 && offset_minutes <= 720) {
00319         timezone_offset_minutes_ = offset_minutes;
00320     } else {
00321         uart_print("Error: Invalid timezone offset", VerbosityLevel::ERROR);
00322     }
00323 }
00324
00325
00334 uint32_t DS3231::get_clock_sync_interval() const {
00335     return sync_interval_minutes_;
00336 }
00337
00338
00349 void DS3231::set_clock_sync_interval(uint32_t interval_minutes) {
00350     if (interval_minutes >= 1 && interval_minutes <= 43200) {
00351         sync_interval_minutes_ = interval_minutes;
00352     } else {
00353         uart_print("Error: Invalid sync interval", VerbosityLevel::ERROR);

```

```

00354     }
00355 }
00356
00357
00367 time_t DS3231::get_last_sync_time() const {
00368     return last_sync_time_;
00369 }
00370
00371
00381 void DS3231::update_last_sync_time() {
00382     last_sync_time_ = get_unix_time();
00383     uart_print("Clock sync time updated: " + std::to_string(last_sync_time_), VerbosityLevel::INFO);
00384 }
00385
00386
00395 time_t DS3231::get_local_time() {
00396     time_t utc_time = get_unix_time();
00397     if (utc_time == -1) {
00398         return -1;
00399     }
00400
00401     return utc_time + (timezone_offset_minutes_ * 60);
00402 }
00403
00404
00415 bool DS3231::is_sync_needed() {
00416     if (last_sync_time_ == 0) {
00417         return true;
00418     }
00419
00420     time_t current_time = get_unix_time();
00421     if (current_time == -1) {
00422         return true;
00423     }
00424
00425     time_t time_since_last_sync = current_time - last_sync_time_;
00426     uint32_t minutes_since_last_sync = time_since_last_sync / 60;
00427
00428     return minutes_since_last_sync >= sync_interval_minutes_;
00429 }
00430
00431
00446 bool DS3231::sync_clock_with_gps() {
00447     auto& nmea_data = NMEADData::get_instance();
00448
00449     if (!nmea_data.has_valid_time()) {
00450         uart_print("GPS time data not available for sync", VerbosityLevel::WARNING);
00451         EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
00452         return false;
00453     }
00454
00455     time_t gps_time = nmea_data.get_unix_time();
00456     if (gps_time <= 0) {
00457         uart_print("Invalid GPS time for sync", VerbosityLevel::ERROR);
00458         EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
00459         return false;
00460     }
00461
00462     if (set_unix_time(gps_time) != 0) {
00463         uart_print("Failed to set system time from GPS", VerbosityLevel::ERROR);
00464         EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC_DATA_NOT_READY);
00465         return false;
00466     }
00467
00468     update_last_sync_time();
00469
00470     EventEmitter::emit(EventGroup::CLOCK, ClockEvent::GPS_SYNC);
00471     uart_print("Clock synced with GPS time: " + std::to_string(gps_time), VerbosityLevel::INFO);
00472
00473     return true;
00474 }
00475
00476 // ===== private methods
00477
00493 int DS3231::i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
00494     if (!length)
00495         return -1;
00496
00497     std::string status = "Reading register " + std::to_string(reg_addr) + " from DS3231";
00498     uart_print(status, VerbosityLevel::DEBUG);
00499     recursive_mutex_enter_blocking(&clock_mutex_);
00500     uint8_t reg = reg_addr;
00501     int write_result = i2c_write_blocking(i2c, ds3231_addr, &reg, 1, true);
00502     if (write_result == PICO_ERROR_GENERIC) {
00503         status = "Failed to write register address to DS3231";
00504         uart_print(status, VerbosityLevel::ERROR);
00505         recursive_mutex_exit(&clock_mutex_);

```

```

00506         return -1;
00507     }
00508     int read_result = i2c_read_blocking(i2c, ds3231_addr, data, length, false);
00509     if (read_result == PICO_ERROR_GENERIC) {
00510         status = "Failed to read register data from DS3231";
00511         uart_print(status, VerbosityLevel::ERROR);
00512         recursive_mutex_exit(&clock_mutex_);
00513         return -1;
00514     }
00515     recursive_mutex_exit(&clock_mutex_);
00516
00517     return 0;
00518 }
00519
00535 int DS3231::i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data) {
00536     if (!length)
00537         return -1;
00538
00539     recursive_mutex_enter_blocking(&clock_mutex_);
00540     std::vector<uint8_t> message(length + 1);
00541     message[0] = reg_addr;
00542     for (size_t i = 0; i < length; i++) {
00543         message[i + 1] = data[i];
00544     }
00545     int write_result = i2c_write_blocking(i2c, ds3231_addr, message.data(), (length + 1), false);
00546     if (write_result == PICO_ERROR_GENERIC) {
00547         uart_print("Error: i2c_write_blocking failed in i2c_write_reg", VerbosityLevel::ERROR);
00548         recursive_mutex_exit(&clock_mutex_);
00549         return -1;
00550     }
00551     recursive_mutex_exit(&clock_mutex_);
00552
00553     return 0;
00554 }
00555
00566 uint8_t DS3231::bin_to_bcd(const uint8_t data) {
00567     uint8_t ones_digit = (uint8_t)(data % 10);
00568     uint8_t tens_digit = (uint8_t)(data - ones_digit) / 10;
00569     return ((tens_digit << 4) + ones_digit);
00570 }
00571
00572
00583 uint8_t DS3231::bcd_to_bin(const uint8_t bcd) {
00584     uint8_t ones_digit = (uint8_t)(bcd & 0x0F);
00585     uint8_t tens_digit = (uint8_t)(bcd >> 4);
00586     return (tens_digit * 10 + ones_digit);
00587 } // End of DS3231_RTC group

```

8.7 lib/clock/DS3231.h File Reference

```

#include <string>
#include <array>
#include "pico/stdlib.h"
#include "hardware/i2c.h"
#include <time.h>
#include "pico/mutex.h"
#include "lib/location/gps_collector.h"

```

Classes

- struct [ds3231_data_t](#)
Structure to hold time and date information from [DS3231](#).
- class [DS3231](#)
Class for interfacing with the [DS3231](#) real-time clock.

Macros

- `#define DS3231_DEVICE_ADRESS 0x68`
DS3231 I2C device address.
- `#define DS3231_SECONDS_REG 0x00`
Register address: Seconds (0-59)
- `#define DS3231_MINUTES_REG 0x01`
Register address: Minutes (0-59)
- `#define DS3231_HOURS_REG 0x02`
Register address: Hours (0-23 in 24hr mode)
- `#define DS3231_DAY_REG 0x03`
Register address: Day of the week (1-7)
- `#define DS3231_DATE_REG 0x04`
Register address: Date (1-31)
- `#define DS3231_MONTH_REG 0x05`
Register address: Month (1-12) & Century bit.
- `#define DS3231_YEAR_REG 0x06`
Register address: Year (00-99)
- `#define DS3231_CONTROL_REG 0x0E`
Register address: Control register.
- `#define DS3231_CONTROL_STATUS_REG 0x0F`
Register address: Control/Status register.
- `#define DS3231_TEMPERATURE_MSB_REG 0x11`
Register address: Temperature register (MSB)
- `#define DS3231_TEMPERATURE_LSB_REG 0x12`
Register address: Temperature register (LSB)

Enumerations

- `enum days_of_week {`
`MONDAY = 1 , TUESDAY , WEDNESDAY , THURSDAY ,`
`FRIDAY , SATURDAY , SUNDAY }`
Enumeration of days of the week.

8.7.1 Macro Definition Documentation

8.7.1.1 DS3231_DEVICE_ADRESS

```
#define DS3231_DEVICE_ADRESS 0x68
```

DS3231 I2C device address.

Definition at line 15 of file [DS3231.h](#).

8.7.1.2 DS3231_SECONDS_REG

```
#define DS3231_SECONDS_REG 0x00
```

Register address: Seconds (0-59)

Definition at line 20 of file [DS3231.h](#).

8.7.1.3 DS3231_MINUTES_REG

```
#define DS3231_MINUTES_REG 0x01
```

Register address: Minutes (0-59)

Definition at line 25 of file [DS3231.h](#).

8.7.1.4 DS3231_HOURS_REG

```
#define DS3231_HOURS_REG 0x02
```

Register address: Hours (0-23 in 24hr mode)

Definition at line 30 of file [DS3231.h](#).

8.7.1.5 DS3231_DAY_REG

```
#define DS3231_DAY_REG 0x03
```

Register address: Day of the week (1-7)

Definition at line 35 of file [DS3231.h](#).

8.7.1.6 DS3231_DATE_REG

```
#define DS3231_DATE_REG 0x04
```

Register address: Date (1-31)

Definition at line 40 of file [DS3231.h](#).

8.7.1.7 DS3231_MONTH_REG

```
#define DS3231_MONTH_REG 0x05
```

Register address: Month (1-12) & Century bit.

Definition at line 45 of file [DS3231.h](#).

8.7.1.8 DS3231_YEAR_REG

```
#define DS3231_YEAR_REG 0x06
```

Register address: Year (00-99)

Definition at line 50 of file [DS3231.h](#).

8.7.1.9 DS3231_CONTROL_REG

```
#define DS3231_CONTROL_REG 0x0E
```

Register address: Control register.

Definition at line 55 of file [DS3231.h](#).

8.7.1.10 DS3231_CONTROL_STATUS_REG

```
#define DS3231_CONTROL_STATUS_REG 0x0F
```

Register address: Control/Status register.

Definition at line 60 of file [DS3231.h](#).

8.7.1.11 DS3231_TEMPERATURE_MSB_REG

```
#define DS3231_TEMPERATURE_MSB_REG 0x11
```

Register address: Temperature register (MSB)

Definition at line 65 of file [DS3231.h](#).

8.7.1.12 DS3231_TEMPERATURE_LSB_REG

```
#define DS3231_TEMPERATURE_LSB_REG 0x12
```

Register address: Temperature register (LSB)

Definition at line 70 of file [DS3231.h](#).

8.7.2 Enumeration Type Documentation

8.7.2.1 days_of_week

```
enum days_of_week
```

Enumeration of days of the week.

Enumerator

MONDAY	Monday.
TUESDAY	Tuesday.
WEDNESDAY	Wednesday.
THURSDAY	Thursday.
FRIDAY	Friday.
SATURDAY	Saturday.
SUNDAY	Sunday.

Definition at line 76 of file [DS3231.h](#).

8.8 DS3231.h

[Go to the documentation of this file.](#)

```

00001 #ifndef DS3231_H
00002 #define DS3231_H
00003
00004 #include <string>
00005 #include <array>
00006 #include "pico/stdlib.h"
00007 #include "hardware/i2c.h"
00008 #include <time.h>
00009 #include "pico/mutex.h"
00010 #include "lib/location/gps_collector.h"
00011
00015 #define DS3231_DEVICE_ADDRESS      0x68
00016
00020 #define DS3231_SECONDS_REG         0x00
00021
00025 #define DS3231_MINUTES_REG         0x01
00026
00030 #define DS3231_HOURS_REG           0x02
00031
00035 #define DS3231_DAY_REG             0x03
00036
00040 #define DS3231_DATE_REG            0x04
00041
00045 #define DS3231_MONTH_REG           0x05
00046
00050 #define DS3231_YEAR_REG            0x06
00051
00055 #define DS3231_CONTROL_REG         0x0E
00056
00060 #define DS3231_CONTROL_STATUS_REG  0x0F
00061
00065 #define DS3231_TEMPERATURE_MSB_REG 0x11
00066
00070 #define DS3231_TEMPERATURE_LSB_REG 0x12
00071
00076 enum days_of_week {
00077     MONDAY    = 1,
00078     TUESDAY,
00079     WEDNESDAY,
00080     THURSDAY,
00081     FRIDAY,
00082     SATURDAY,
00083     SUNDAY
00084 };
00085
00090 typedef struct {
00091     uint8_t seconds;
00092     uint8_t minutes;
00093     uint8_t hours;
00094     uint8_t day;
00095     uint8_t date;
00096     uint8_t month;
00097     uint8_t year;
00098     bool century;
00099 } ds3231_data_t;
00100
00108 class DS3231 {
00109 public:
00115     DS3231(i2c_inst_t *i2c_instance);
00121     static DS3231& get_instance();
00122
00129     int set_time(ds3231_data_t *data);
00130
00137     int get_time(ds3231_data_t *data);
00138
00145     int read_temperature(float *resolution);
00146
00153     int set_unix_time(time_t unix_time);
00154
00160     time_t get_unix_time();
00161
00167     int clock_enable();
00168
00174     int16_t get_timezone_offset() const;
00175
00181     void set_timezone_offset(int16_t offset_minutes);
00182
00188     uint32_t get_clock_sync_interval() const;
00189
00195     void set_clock_sync_interval(uint32_t interval_minutes);
00196
00202     time_t get_last_sync_time() const;

```

```

00203
00207     void update_last_sync_time();
00208
00214     time_t get_local_time();
00215
00221     bool is_sync_needed();
00222
00228     bool sync_clock_with_gps();
00229
00230
00231 private:
00232     i2c_inst_t *i2c;
00233     uint8_t ds3231_addr;
00234     recursive_mutex_t clock_mutex;
00235     int16_t timezone_offset_minutes_ = 60;
00236     uint32_t sync_interval_minutes_ = 1440;
00237     time_t last_sync_time_ = 0;
00238
00239     // Private constructor
00240     DS3231();
00241
00242     // Delete copy constructor and assignment operator
00243     DS3231(const DS3231&) = delete;
00244     DS3231& operator=(const DS3231&) = delete;
00245
00254     int i2c_read_reg(uint8_t reg_addr, size_t length, uint8_t *data);
00255
00264     int i2c_write_reg(uint8_t reg_addr, size_t length, uint8_t *data);
00265
00272     uint8_t bin_to_bcd(const uint8_t data);
00273
00280     uint8_t bcd_to_bin(const uint8_t bcd);
00281 };
00282
00283 #endif // DS3231_H

```

8.9 lib/comms/commands/clock_commands.cpp File Reference

```

#include "communication.h"
#include <time.h>
#include "DS3231.h"

```

Macros

- #define [CLOCK_GROUP](#) 3
- #define [TIME](#) 0
- #define [TIMEZONE_OFFSET](#) 1
- #define [CLOCK_SYNC_INTERVAL](#) 2
- #define [LAST_SYNC_TIME](#) 3

Functions

- std::vector< [Frame](#) > [handle_time](#) (const std::string ¶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.

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
00010
00016
00017
00031 std::vector<Frame> handle_time(const std::string& param, OperationType operationType) {
00032     std::vector<Frame> frames;
00033     std::string error_msg;
00034
00035     if (operationType == OperationType::SET) {
00036         if (param.empty()) {
00037             error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00038             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00039             return frames;
00040         }
00041         try {
00042             time_t newTime = std::stoll(param);
00043             if (newTime <= 0) {
00044                 error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00045                 frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00046                 return frames;
00047             }
00048
00049             if (DS3231::get_instance().set_unix_time(newTime) != 0) {
00050                 error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
00051                 frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00052                 return frames;
00053             }
00054
00055             EventEmitter::emit(EventGroup::CLOCK, ClockEvent::CHANGED);
00056             frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, TIME,
std::to_string(DS3231::get_instance().get_unix_time())));
00057             return frames;
00058         } catch (...) {
00059             error_msg = error_code_to_string(ErrorCode::INVALID_FORMAT);
00060             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00061             return frames;
00062         }
00063     } else if (operationType == OperationType::GET) {
00064         if (!param.empty()) {
00065             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00066             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00067             return frames;
00068         }
00069
00070         uint32_t time_unix = DS3231::get_instance().get_local_time();
00071         if (time_unix == 0) {
00072             error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
00073             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00074             return frames;
00075         }
00076
00077         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, TIME,
std::to_string(time_unix)));
00078         return frames;
00079     }
00080
00081     error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00082     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIME, error_msg));
00083     return frames;
00084 }
00085
00096 std::vector<Frame> handle_timezone_offset(const std::string& param, OperationType operationType) {
00097     std::vector<Frame> frames;
00098     std::string error_msg;
00099
00100     if (!(operationType == OperationType::GET || operationType == OperationType::SET)) {
00101         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00102         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
00103         return frames;
00104     }
00105
00106     if (operationType == OperationType::GET) {
00107         if (!param.empty()) {
00108             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);

```

```

00109         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET,
error_msg));
00110         return frames;
00111     }
00112
00113     int offset = DS3231::get_instance().get_timezone_offset();
00114     std::string offset_set = std::to_string(offset);
00115     frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, TIMEZONE_OFFSET, offset_set));
00116     return frames;
00117 }
00118
00119 if (param.empty()) {
00120     error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00121     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
00122     return frames;
00123 }
00124
00125 try {
00126     int16_t offset = std::stoi(param);
00127     if (offset < -720 || offset > 720) {
00128         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00129         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET,
error_msg));
00130         return frames;
00131     }
00132
00133     DS3231::get_instance().set_timezone_offset(offset);
00134     std::string offset_set = std::to_string(offset);
00135     frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, TIMEZONE_OFFSET, offset_set));
00136     return frames;
00137 } catch (...) {
00138     error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00139     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, TIMEZONE_OFFSET, error_msg));
00140     return frames;
00141 }
00142 }
00143
00144
00155 std::vector<Frame> handle_clock_sync_interval(const std::string& param, OperationType operationType) {
00156     std::vector<Frame> frames;
00157     std::string error_msg;
00158
00159     if (! (operationType == OperationType::GET || operationType == OperationType::SET)) {
00160         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00161         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
error_msg));
00162         return frames;
00163     }
00164
00165     if (operationType == OperationType::GET) {
00166         if (!param.empty()) {
00167             error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00168             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
error_msg));
00169             return frames;
00170         }
00171
00172         uint32_t syncInterval = DS3231::get_instance().get_clock_sync_interval();
00173         std::string clockSyncInterval = std::to_string(syncInterval);
00174         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
clockSyncInterval));
00175         return frames;
00176     }
00177
00178     if (operationType == OperationType::SET) {
00179         if (param.empty()) {
00180             error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00181             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
error_msg));
00182             return frames;
00183         }
00184         try {
00185             uint32_t interval = std::stoul(param);
00186
00187             DS3231::get_instance().set_clock_sync_interval(interval);
00188             std::string interval_set = std::to_string(interval);
00189
00190             frames.push_back(frame_build(OperationType::RES, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
interval_set));
00191             return frames;
00192         } catch (...) {
00193             error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00194             frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL,
error_msg));
00195         }
00196     }
00197     error_msg = error_code_to_string(ErrorCode::UNKNOWN_ERROR);

```

```

00198     frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, CLOCK_SYNC_INTERVAL, error_msg));
00199     return frames;
00200 }
00201
00211 std::vector<Frame> handle_get_last_sync_time(const std::string& param, OperationType operationType) {
00212     std::vector<Frame> frames;
00213     std::string error_msg;
00214
00215     if (operationType != OperationType::GET || !param.empty()) {
00216         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00217         frames.push_back(frame_build(OperationType::ERR, CLOCK_GROUP, LAST_SYNC_TIME, error_msg));
00218         return frames;
00219     }
00220
00221     time_t lastSyncTime = DS3231::get_instance().get_last_sync_time();
00222
00223     if (lastSyncTime == 0) {
00224         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, LAST_SYNC_TIME, "NEVER"));
00225     } else {
00226         frames.push_back(frame_build(OperationType::VAL, CLOCK_GROUP, LAST_SYNC_TIME,
00227                                     std::to_string(lastSyncTime)));
00228     }
00229
00230     return frames;
00231 } // end of ClockCommands group

```

8.11 lib/comms/commands/commands.cpp File Reference

```

#include "commands.h"
#include "communication.h"

```

Typedefs

- using [CommandHandler](#) = std::function<std::vector<[Frame](#)>(const std::string&, [OperationType](#))>
Function type for command handlers.
- using [CommandMap](#) = std::map<uint32_t, [CommandHandler](#)>
Map type for storing command handlers.

Functions

- std::vector< [Frame](#) > [execute_command](#) (uint32_t commandKey, const std::string ¶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
00029     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(1)), handle_get_build_version},
00030     // Group 1, Command 1
00030     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(8)), handle_verbosity},
00031     // Group 1, Command 9
00031     {((static_cast<uint32_t>(1) << 8) | static_cast<uint32_t>(9)), handle_enter_bootloader_mode},
00032     // Group 2, Command 0
00032     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(0)), handle_get_power_manager_ids},
00033     // Group 2, Command 2
00033     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(2)), handle_get_voltage_battery},
00034     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(3)), handle_get_voltage_5v},
00035     // Group 2, Command 3
00035     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(4)), handle_get_current_charge_usb},
00036     // Group 2, Command 4
00036     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(5)), handle_get_current_charge_solar},
00037     // Group 2, Command 5
00037     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(6)), handle_get_current_charge_total},
00038     // Group 2, Command 6
00038     {((static_cast<uint32_t>(2) << 8) | static_cast<uint32_t>(7)), handle_get_current_draw},
00039     // Group 2, Command 7
00039     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(0)), handle_time},
00040     // Group 3, Command 0
00040     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(1)), handle_timezone_offset},
00041     // Group 3, Command 1
00041     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(2)), handle_clock_sync_interval},
00042     // Group 3, Command 2
00042     {((static_cast<uint32_t>(3) << 8) | static_cast<uint32_t>(3)), handle_get_last_sync_time},
00043     // Group 3, Command 3
00043     {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(0)), handle_get_sensor_data},
00044     // Group 4, Command 0
00044     {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(1)), handle_sensor_config},
00045     // Group 4, Command 1
00045     {((static_cast<uint32_t>(4) << 8) | static_cast<uint32_t>(3)), handle_get_sensor_list},
00046     // Group 4, Command 3
00046     {((static_cast<uint32_t>(5) << 8) | static_cast<uint32_t>(1)), handle_get_last_events},
00047     // Group 5, Command 1
00047     {((static_cast<uint32_t>(5) << 8) | static_cast<uint32_t>(2)), handle_get_event_count},
00048     // Group 5, Command 2
00048     {((static_cast<uint32_t>(6) << 8) | static_cast<uint32_t>(0)), handle_list_files},
00049     // Group 6, Command 0
00049     {((static_cast<uint32_t>(6) << 8) | static_cast<uint32_t>(4)), handle_mount},
00050     // Group 6, Command 4
00050     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(1)), handle_gps_power_status},
00051     // Group 7, Command 1
00051     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(2)), handle_enable_gps_uart_passthrough},
00052     // Group 7, Command 3
00052     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(3)), handle_get_rmc_data},
00053     // Group 7, Command 3
00053     {((static_cast<uint32_t>(7) << 8) | static_cast<uint32_t>(4)), handle_get_gga_data},
00054     // Group 7, Command 4
00054     {((static_cast<uint32_t>(8) << 8) | static_cast<uint32_t>(2)), handle_get_last_telemetry_record},
00055     {((static_cast<uint32_t>(8) << 8) | static_cast<uint32_t>(3)), handle_get_last_sensor_record},
00056 };
00057
00058
00067 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
operationType) {
00068     auto it = command_handlers.find(commandKey);
00069     if (it != command_handlers.end()) {
00070         CommandHandler handler = it->second;
00071         return handler(param, operationType);
00072     } else {
00073         std::vector<Frame> frames;
00074         frames.push_back(frame_build(OperationType::ERR, 0, 0, "INVALID COMMAND"));
00075         return frames;
00076     }
00077 } // end of CommandSystem group

```

8.13 lib/comms/commands/commands.h File Reference

```
#include <string>
#include <functional>
#include <map>
#include "protocol.h"
```

Functions

- `std::vector< Frame > handle_time` (const `std::string` ¶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)
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.
- `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
operationType);

```

```

00022
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
operationType);
00027 std::vector<Frame> handle_get_rmc_data(const std::string& param, OperationType operationType);
00028 std::vector<Frame> handle_get_gga_data(const std::string& param, OperationType operationType);
00029
00030
00031 // POWER
00032 std::vector<Frame> handle_get_power_manager_ids(const std::string& param, OperationType
operationType);
00033 std::vector<Frame> handle_get_voltage_battery(const std::string& param, OperationType operationType);
00034 std::vector<Frame> handle_get_voltage_5v(const std::string& param, OperationType operationType);
00035 std::vector<Frame> handle_get_current_charge_usb(const std::string& param, OperationType
operationType);
00036 std::vector<Frame> handle_get_current_charge_solar(const std::string& param, OperationType
operationType);
00037 std::vector<Frame> handle_get_current_charge_total(const std::string& param, OperationType
operationType);
00038 std::vector<Frame> handle_get_current_draw(const std::string& param, OperationType operationType);
00039
00040
00041 // EVENT
00042 std::vector<Frame> handle_get_last_events(const std::string& param, OperationType operationType);
00043 std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType);
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
operationType);
00058 std::vector<Frame> handle_get_last_sensor_record(const std::string& param, OperationType
operationType);
00059
00060 std::vector<Frame> execute_command(uint32_t commandKey, const std::string& param, OperationType
operationType);
00061 extern std::map<uint32_t, std::function<std::vector<Frame>(const std::string&, OperationType)>>
command_handlers;
00062
00063 #endif

```

8.15 lib/comms/commands/diagnostic_commands.cpp File Reference

```

#include "communication.h"
#include "commands.h"
#include "pico/stdlib.h"
#include "pico/bootrom.h"
#include "system_state_manager.h"

```

Functions

- `std::vector< Frame > handle_get_commands_list` (const std::string ¶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         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 {
00159     std::vector<Frame> frames;
00160     std::string error_msg;
00161
00162     if (param != "USB") {
00163         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00164         frames.push_back(frame_build(OperationType::ERR, 1, 9, error_msg));
00165         return frames;
00166     }
00167
00168     if (operationType != OperationType::SET) {
00169         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00170         frames.push_back(frame_build(OperationType::ERR, 1, 9, error_msg));
00171         return frames;
00172     }
00173
00174     frames.push_back(frame_build(OperationType::RES, 1, 9, "REBOOT BOOTSEL"));
00175
00176     SystemStateManager::get_instance().set_bootloader_reset_pending(true);
00177
00178     return frames;
00179 }

```

8.17 lib/comms/commands/event_commands.cpp File Reference

```

#include "communication.h"
#include "event_manager.h"
#include <sstream>

```

Functions

- `std::vector< Frame > handle_get_last_events` (const std::string ¶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
00043     size_t count = 10; // Default number of events to return
00044     if (!param.empty()) {
00045         try {
00046             count = std::stoul(param);
00047             if (count > EVENT_BUFFER_SIZE) {
00048                 error_msg = error_code_to_string(ErrorCode::INVALID_VALUE);
00049                 frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
00050                 return frames;
00051             }
00052         } catch (...) {
00053             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00054             frames.push_back(frame_build(OperationType::ERR, 5, 1, error_msg));
00055             return frames;
00056         }
00057     }
00058
00059     auto& event_manager = EventManager::get_instance();
00060     size_t available = event_manager.get_event_count();
00061     size_t to_return = (count == 0) ? available : std::min(count, available);
00062     size_t event_index = available;
00063
00064     while (to_return > 0) {
00065         std::stringstream ss;
00066         ss << std::hex << std::uppercase << std::setfill('0');
00067         size_t events_in_frame = 0;
00068
00069         for (size_t i = 0; i < 10 && to_return > 0; ++i) {
00070             event_index--;
00071             const EventLog& event = event_manager.get_event(event_index);
00072
00073             ss << std::setw(4) << event.id
00074                << std::setw(8) << event.timestamp
00075                << std::setw(2) << static_cast<int>(event.group)
00076                << std::setw(2) << static_cast<int>(event.event);
00077
00078             if (to_return > 1) ss << "-";
00079             to_return--;
00080             events_in_frame++;
00081         }
00082         frames.push_back(frame_build(OperationType::SEQ, 5, 1, ss.str()));
00083     }
00084     frames.push_back(frame_build(OperationType::VAL, 5, 1, "SEQ_DONE"));
00085     return frames;
00086 }
00087
00088
00101 std::vector<Frame> handle_get_event_count(const std::string& param, OperationType operationType) {
00102     std::vector<Frame> frames;
00103     std::string error_msg;
00104
00105     if (operationType != OperationType::GET || !param.empty()) {
00106         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00107         frames.push_back(frame_build(OperationType::ERR, 5, 2, error_msg));
00108         return frames;
00109     }
00110
00111     auto& event_manager = EventManager::get_instance();
00112     frames.push_back(frame_build(OperationType::VAL, 5, 2,
00113         std::to_string(event_manager.get_event_count())));
00114     return frames;
00115 } // end of EventCommands group

```

8.19 lib/comms/commands/gps_commands.cpp File Reference

```

#include "communication.h"
#include "lib/location/gps_collector.h"
#include <sstream>
#include "system_state_manager.h"

```

Macros

- `#define GPS_GROUP 7`
- `#define POWER_STATUS_COMMAND 1`
- `#define PASSTHROUGH_COMMAND 2`
- `#define RMC_DATA_COMMAND 3`
- `#define GGA_DATA_COMMAND 4`

Functions

- `std::vector< Frame > handle_gps_power_status` (const std::string ¶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
00010 #define GGA_DATA_COMMAND 4
00011
00017
00033 std::vector<Frame> handle_gps_power_status(const std::string& param, OperationType operationType) {
00034     std::vector<Frame> frames;
00035     std::string error_str;
00036
00037     if (operationType == OperationType::SET) {
00038         if (param.empty()) {
00039             error_str = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00040             frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
error_str));
00041             return frames;
00042         }
00043
00044         try {
00045             int power_status = std::stoi(param);
00046             if (power_status != 0 && power_status != 1) {
00047                 error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
00048                 frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
error_str));
00049                 return frames;
00050             }
00051             gpio_put(GPS_POWER_ENABLE_PIN, power_status);
00052             EventEmitter::emit(EventGroup::GPS, power_status ? GPSEvent::POWER_ON :
GPSEvent::POWER_OFF);
00053             frames.push_back(frame_build(OperationType::RES, GPS_GROUP, POWER_STATUS_COMMAND,
std::to_string(power_status)));
00054             return frames;
00055         } catch (...) {
00056             error_str = error_code_to_string(ErrorCode::PARAM_INVALID);
00057             frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND,
error_str));
00058             return frames;
00059         }
00060     }
00061
00062     // GET operation
00063     if (!param.empty()) {
00064         error_str = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00065         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, POWER_STATUS_COMMAND, error_str));
00066         return frames;
00067     }
00068
00069     bool power_status = gpio_get(GPS_POWER_ENABLE_PIN);
00070     frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, POWER_STATUS_COMMAND,
std::to_string(power_status)));
00071     return frames;
00072 }
00073
00074
00090 std::vector<Frame> handle_enable_gps_uart_passthrough(const std::string& param, OperationType
operationType) {
00091     std::vector<Frame> frames;
00092     std::string error_str;
00093
00094     if (!(operationType == OperationType::SET)) {
00095         error_str = error_code_to_string(ErrorCode::INVALID_OPERATION);
00096         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
00097         return frames;
00098     }
00099 }
```

```

00098     }
00099
00100     // Parse and validate timeout parameter
00101     uint32_t timeout_ms;
00102     try {
00103         timeout_ms = param.empty() ? 60000u : std::stoul(param) * 1000;
00104     } catch (...) {
00105         error_str = error_code_to_string(ErrorCode::INVALID_VALUE);
00106         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, PASSTHROUGH_COMMAND, error_str));
00107         return frames;
00108     }
00109
00110     // 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
00129     sleep_ms(10);
00130
00131     // Change main UART baudrate to GPS module baudrate for passthrough duration
00132     uart_set_baudrate(DEBUG_UART_PORT, gps_baud_rate);
00133
00134     while (!exit_requested) {
00135         while (uart_is_readable(DEBUG_UART_PORT)) {
00136             char ch = uart_getc(DEBUG_UART_PORT);
00137
00138             input_buffer += ch;
00139             if (input_buffer.length() > EXIT_SEQUENCE.length()) {
00140                 input_buffer = input_buffer.substr(1);
00141             }
00142
00143             if (input_buffer == EXIT_SEQUENCE) {
00144                 exit_requested = true;
00145                 break;
00146             }
00147
00148             if (input_buffer != EXIT_SEQUENCE.substr(0, input_buffer.length())) {
00149                 uart_write_blocking(GPS_UART_PORT,
00150                                   reinterpret_cast<const uint8_t*>(&ch), 1);
00151             }
00152         }
00153
00154         while (uart_is_readable(GPS_UART_PORT)) {
00155             char gps_byte = uart_getc(GPS_UART_PORT);
00156             uart_write_blocking(DEBUG_UART_PORT,
00157                               reinterpret_cast<const uint8_t*>(&gps_byte), 1);
00158         }
00159
00160         if (to_ms_since_boot(get_absolute_time()) - start_time >= timeout_ms) {
00161             break;
00162         }
00163     }
00164
00165     uart_set_baudrate(DEBUG_UART_PORT, original_baud_rate);
00166
00167     sleep_ms(50);
00168
00169     SystemStateManager::get_instance().set_gps_collection_paused(false);
00170     EventEmitter::emit(EventGroup::GPS, GPSEvent::PASS_THROUGH_END);
00171
00172     std::string exit_reason = exit_requested ? "USER_EXIT" : "TIMEOUT";
00173     std::string response = "GPS UART BRIDGE EXIT: " + exit_reason;
00174     uart_print(response, VerbosityLevel::INFO);
00175
00176     frames.push_back(frame_build(OperationType::RES, GPS_GROUP, PASSTHROUGH_COMMAND, response));
00177     return frames;
00178 }
00179
00180
00193 std::vector<Frame> handle_get_rmc_data(const std::string& param, OperationType operationType) {
00194     std::vector<Frame> frames;
00195     std::string error_msg;
00196

```



```

00197     if (operationType != OperationType::GET) {
00198         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00199         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, RMC_DATA_COMMAND, error_msg));
00200         return frames;
00201     }
00202
00203     if (!param.empty()) {
00204         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00205         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, RMC_DATA_COMMAND, error_msg));
00206         return frames;
00207     }
00208
00209     auto& nmea_data = NMEADData::get_instance();
00210     std::vector<std::string> tokens = nmea_data.get_rmc_tokens();
00211     std::stringstream ss;
00212     for (size_t i = 0; i < tokens.size(); ++i) {
00213         ss << tokens[i];
00214         if (i < tokens.size() - 1) {
00215             ss << ",";
00216         }
00217     }
00218
00219     frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, RMC_DATA_COMMAND, ss.str()));
00220     return frames;
00221 }
00222
00223
00236 std::vector<Frame> handle_get_gga_data(const std::string& param, OperationType operationType) {
00237     std::vector<Frame> frames;
00238     std::string error_msg;
00239
00240     if (operationType != OperationType::GET) {
00241         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00242         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, GGA_DATA_COMMAND, error_msg));
00243         return frames;
00244     }
00245
00246     if (!param.empty()) {
00247         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00248         frames.push_back(frame_build(OperationType::ERR, GPS_GROUP, GGA_DATA_COMMAND, error_msg));
00249         return frames;
00250     }
00251
00252     auto& nmea_data = NMEADData::get_instance();
00253     std::vector<std::string> tokens = nmea_data.get_gga_tokens();
00254     std::stringstream ss;
00255     for (size_t i = 0; i < tokens.size(); ++i) {
00256         ss << tokens[i];
00257         if (i < tokens.size() - 1) {
00258             ss << ",";
00259         }
00260     }
00261
00262     frames.push_back(frame_build(OperationType::VAL, GPS_GROUP, GGA_DATA_COMMAND, ss.str()));
00263     return frames;
00264 } // end of GPSCommands group

```

8.21 lib/comms/commands/power_commands.cpp File Reference

```
#include "communication.h"
```

Macros

- #define [POWER_GROUP](#) 2
- #define [POWER_MANAGER_IDS](#) 0
- #define [VOLTAGE_BATTERY](#) 2
- #define [VOLTAGE_MAIN](#) 3
- #define [CHARGE_USB](#) 4
- #define [CHARGE_SOLAR](#) 5
- #define [CHARGE_TOTAL](#) 6
- #define [DRAW_TOTAL](#) 7

Functions

- `std::vector< Frame > handle_get_power_manager_ids (const std::string ¶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
00017
00030 std::vector<Frame> handle_get_power_manager_ids(const std::string& param, OperationType operationType)
00031 {
00032     std::vector<Frame> frames;
00033     std::string error_msg;
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     if ((operationType == OperationType::GET)) {
00040         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00041         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, POWER_MANAGER_IDS, error_msg));
00042         return frames;
00043     }
00044     std::string power_manager_ids = PowerManager::get_instance().read_device_ids();
00045     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, POWER_MANAGER_IDS,
00046         power_manager_ids));
00047     return frames;
00048 }
00049

```

```

00050
00063 std::vector<Frame> handle_get_voltage_battery(const std::string& param, OperationType operationType) {
00064     std::vector<Frame> frames;
00065     std::string error_msg;
00066
00067     if (!param.empty()) {
00068         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00069         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_BATTERY, error_msg));
00070         return frames;
00071     }
00072
00073     if (!(operationType == OperationType::GET)) {
00074         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00075         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_BATTERY, error_msg));
00076         return frames;
00077     }
00078
00079     float voltage = PowerManager::get_instance().get_voltage_battery();
00080     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, VOLTAGE_BATTERY,
std::to_string(voltage), ValueUnit::VOLT));
00081     return frames;
00082 }
00083
00096 std::vector<Frame> handle_get_voltage_5v(const std::string& param, OperationType operationType) {
00097     std::vector<Frame> frames;
00098     std::string error_msg;
00099
00100     if (!param.empty()) {
00101         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00102         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_MAIN, error_msg));
00103         return frames;
00104     }
00105
00106     if (!(operationType == OperationType::GET)) {
00107         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00108         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, VOLTAGE_MAIN, error_msg));
00109         return frames;
00110     }
00111
00112     float voltage = PowerManager::get_instance().get_voltage_5v();
00113     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, VOLTAGE_MAIN,
std::to_string(voltage), ValueUnit::VOLT));
00114     return frames;
00115 }
00116
00129 std::vector<Frame> handle_get_current_charge_usb(const std::string& param, OperationType
operationType) {
00130     std::vector<Frame> frames;
00131     std::string error_msg;
00132
00133     if (!param.empty()) {
00134         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00135         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_USB, error_msg));
00136         return frames;
00137     }
00138
00139     if (!(operationType == OperationType::GET)) {
00140         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00141         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_USB, error_msg));
00142         return frames;
00143     }
00144
00145     float charge_current = PowerManager::get_instance().get_current_charge_usb();
00146     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_USB,
std::to_string(charge_current), ValueUnit::MILIAMP));
00147     return frames;
00148 }
00149
00162 std::vector<Frame> handle_get_current_charge_solar(const std::string& param, OperationType
operationType) {
00163     std::vector<Frame> frames;
00164     std::string error_msg;
00165
00166     if (!param.empty()) {
00167         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00168         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_SOLAR, error_msg));
00169         return frames;
00170     }
00171
00172     if (!(operationType == OperationType::GET)) {
00173         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00174         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_SOLAR, error_msg));
00175         return frames;
00176     }
00177
00178     float charge_current = PowerManager::get_instance().get_current_charge_solar();
00179     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_SOLAR,

```

```

        std::to_string(charge_current), ValueUnit::MILIAMP));
00180     return frames;
00181 }
00182
00195 std::vector<Frame> handle_get_current_charge_total(const std::string& param, OperationType
operationType) {
00196     std::vector<Frame> frames;
00197     std::string error_msg;
00198
00199     if (!param.empty()) {
00200         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00201         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_TOTAL, error_msg));
00202         return frames;
00203     }
00204
00205     if (!(operationType == OperationType::GET)) {
00206         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00207         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, CHARGE_TOTAL, error_msg));
00208         return frames;
00209     }
00210
00211     float charge_current = PowerManager::get_instance().get_current_charge_total();
00212     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, CHARGE_TOTAL,
std::to_string(charge_current), ValueUnit::MILIAMP));
00213     return frames;
00214 }
00215
00228 std::vector<Frame> handle_get_current_draw(const std::string& param, OperationType operationType) {
00229     std::vector<Frame> frames;
00230     std::string error_msg;
00231
00232     if (!param.empty()) {
00233         error_msg = error_code_to_string(ErrorCode::PARAM_UNNECESSARY);
00234         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, DRAW_TOTAL, error_msg));
00235         return frames;
00236     }
00237
00238     if (!(operationType == OperationType::GET)) {
00239         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00240         frames.push_back(frame_build(OperationType::ERR, POWER_GROUP, DRAW_TOTAL, error_msg));
00241         return frames;
00242     }
00243
00244     float current_draw = PowerManager::get_instance().get_current_draw();
00245     frames.push_back(frame_build(OperationType::VAL, POWER_GROUP, DRAW_TOTAL,
std::to_string(current_draw), ValueUnit::MILIAMP));
00246     return frames;
00247 } // end of PowerCommands group

```

8.23 lib/comms/commands/sensor_commands.cpp File Reference

```

#include "communication.h"
#include "ISensor.h"
#include <vector>
#include <string>
#include <sstream>
#include "commands.h"

```

Macros

- #define [SENSOR_GROUP](#) 4
- #define [SENSOR_READ](#) 0
- #define [SENSOR_CONFIGURE](#) 1

Functions

- std::vector< [Frame](#) > [handle_get_sensor_data](#) (const std::string ¶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 <sstream>
00006 #include "commands.h"
00007
00008 #define SENSOR_GROUP 4
00009 #define SENSOR_READ 0
00010 #define SENSOR_CONFIGURE 1
00011
00017
00034 std::vector<Frame> handle\_get\_sensor\_data(const std::string& param, OperationType operationType) {
00035     std::vector<Frame> frames;
00036     std::string error_msg;
00037
00038     if (param.empty()) {
00039         error_msg = error\_code\_to\_string(ErrorCode::PARAM\_REQUIRED);
00040         frames.push_back(frame\_build(OperationType::ERR, SENSOR\_GROUP, SENSOR\_READ, error_msg));
00041         return frames;
00042     }
00043
00044     if (operationType != OperationType::GET) {
00045         error_msg = error\_code\_to\_string(ErrorCode::INVALID\_OPERATION);
00046         frames.push_back(frame\_build(OperationType::ERR, SENSOR\_GROUP, SENSOR\_READ, error_msg));
00047         return frames;
00048     }
00049
00050     // Parse sensor type and data type from param
00051     std::string sensor_type_str;
00052     std::string data_type_str;
00053
```

```

00054     size_t dash_pos = param.find('-');
00055     if (dash_pos != std::string::npos) {
00056         sensor_type_str = param.substr(0, dash_pos);
00057         data_type_str = param.substr(dash_pos + 1);
00058     } else {
00059         sensor_type_str = param;
00060     }
00061
00062     // Convert sensor_type_str to SensorType
00063     SensorType sensor_type;
00064     if (sensor_type_str == "light") {
00065         sensor_type = SensorType::LIGHT;
00066     } else if (sensor_type_str == "environment") {
00067         sensor_type = SensorType::ENVIRONMENT;
00068     } else {
00069         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid sensor type";
00070         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
00071         return frames;
00072     }
00073
00074     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00075
00076     // If data type is specified, read that specific data
00077     if (!data_type_str.empty()) {
00078         SensorDataTypeIdentifier data_type;
00079
00080         // Map string to SensorDataTypeIdentifier
00081         if (data_type_str == "light_level") {
00082             data_type = SensorDataTypeIdentifier::LIGHT_LEVEL;
00083         } else if (data_type_str == "temperature") {
00084             data_type = SensorDataTypeIdentifier::TEMPERATURE;
00085         } else if (data_type_str == "pressure") {
00086             data_type = SensorDataTypeIdentifier::PRESSURE;
00087         } else if (data_type_str == "humidity") {
00088             data_type = SensorDataTypeIdentifier::HUMIDITY;
00089         } else {
00090             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid data type";
00091             frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_READ, error_msg));
00092             return frames;
00093         }
00094
00095         float value = sensor_wrapper.sensor_read_data(sensor_type, data_type);
00096         std::stringstream ss;
00097         ss << value;
00098         frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, SENSOR_READ, ss.str()));
00099     }
00100     // If only sensor type is specified, read all relevant data for that sensor type
00101     else {
00102         std::vector<SensorDataTypeIdentifier> data_types;
00103
00104         switch (sensor_type) {
00105             case SensorType::LIGHT:
00106                 data_types = {SensorDataTypeIdentifier::LIGHT_LEVEL};
00107                 break;
00108             case SensorType::ENVIRONMENT:
00109                 data_types = {
00110                     SensorDataTypeIdentifier::TEMPERATURE,
00111                     SensorDataTypeIdentifier::PRESSURE,
00112                     SensorDataTypeIdentifier::HUMIDITY
00113                 };
00114                 break;
00115             default:
00116                 break;
00117         }
00118
00119         std::stringstream combined_values;
00120         std::vector<std::string> data_type_names;
00121         std::vector<float> values;
00122
00123         // Get names for the data types and store the values
00124         for (SensorDataTypeIdentifier data_type : data_types) {
00125             switch (data_type) {
00126                 case SensorDataTypeIdentifier::LIGHT_LEVEL:
00127                     data_type_names.push_back("light_level");
00128                     break;
00129                 case SensorDataTypeIdentifier::TEMPERATURE:
00130                     data_type_names.push_back("temperature");
00131                     break;
00132                 case SensorDataTypeIdentifier::PRESSURE:
00133                     data_type_names.push_back("pressure");
00134                     break;
00135                 case SensorDataTypeIdentifier::HUMIDITY:
00136                     data_type_names.push_back("humidity");
00137                     break;
00138                 default:
00139                     break;
00140             }

```

```

00141         float value = sensor_wrapper.sensor_read_data(sensor_type, data_type);
00142         values.push_back(value);
00143     }
00144 }
00145
00146 // Format output as key-value pairs
00147 for (size_t i = 0; i < data_type_names.size(); i++) {
00148     if (i > 0) combined_values « "|";
00149     combined_values « data_type_names[i] « ":" « values[i];
00150 }
00151
00152 frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, SENSOR_READ,
combined_values.str()));
00153 }
00154
00155 return frames;
00156 }
00157
00172 std::vector<Frame> handle_sensor_config(const std::string& param, OperationType operationType) {
00173     std::vector<Frame> frames;
00174     std::string error_msg;
00175
00176     if (param.empty()) {
00177         error_msg = error_code_to_string(ErrorCode::PARAM_REQUIRED);
00178         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
00179         return frames;
00180     }
00181
00182     if (operationType != OperationType::SET) {
00183         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00184         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
00185         return frames;
00186     }
00187
00188     // Parse sensor type and configuration from param
00189     size_t semicolon_pos = param.find(';');
00190     if (semicolon_pos == std::string::npos) {
00191         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Format should be
sensor_type;config_params";
00192         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
00193         return frames;
00194     }
00195
00196     std::string sensor_type_str = param.substr(0, semicolon_pos);
00197     std::string config_str = param.substr(semicolon_pos + 1);
00198
00199     // Convert sensor_type_str to SensorType
00200     SensorType sensor_type;
00201     if (sensor_type_str == "light") {
00202         sensor_type = SensorType::LIGHT;
00203     } else if (sensor_type_str == "environment") {
00204         sensor_type = SensorType::ENVIRONMENT;
00205     } else {
00206         error_msg = error_code_to_string(ErrorCode::PARAM_INVALID) + ": Invalid sensor type";
00207         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
00208         return frames;
00209     }
00210
00211     // Parse configuration parameters
00212     std::map<std::string, std::string> config_map;
00213     std::stringstream ss(config_str);
00214     std::string config_pair;
00215
00216     while (std::getline(ss, config_pair, '|')) {
00217         size_t colon_pos = config_pair.find(':');
00218         if (colon_pos != std::string::npos) {
00219             std::string key = config_pair.substr(0, colon_pos);
00220             std::string value = config_pair.substr(colon_pos + 1);
00221             config_map[key] = value;
00222         }
00223     }
00224
00225     // Apply configuration
00226     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00227     bool success = sensor_wrapper.sensor_configure(sensor_type, config_map);
00228
00229     if (success) {
00230         frames.push_back(frame_build(OperationType::RES, SENSOR_GROUP, SENSOR_CONFIGURE,
"Configuration successful"));
00231     } else {
00232         error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET) + ": Failed to configure sensor";
00233         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, SENSOR_CONFIGURE, error_msg));
00234     }
00235
00236     return frames;
00237 }
00238

```



```

00239
00252 std::vector<Frame> handle_get_sensor_list([[maybe_unused]] const std::string& param, OperationType
operationType) {
00253     std::vector<Frame> frames;
00254     std::string error_msg;
00255
00256     if (operationType != OperationType::GET) {
00257         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00258         frames.push_back(frame_build(OperationType::ERR, SENSOR_GROUP, 2, error_msg));
00259         return frames;
00260     }
00261
00262     // Get the singleton instance
00263     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00264
00265     // Get list of available sensor types
00266     std::vector<std::pair<SensorType, uint8_t> > available_sensors =
sensor_wrapper.get_available_sensors();
00267
00268     if (available_sensors.empty()) {
00269         frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, 2, "No sensors available"));
00270         return frames;
00271     }
00272
00273     std::stringstream sensor_list;
00274     bool first = true;
00275
00276     for (const auto& sensor_info : available_sensors) {
00277         if (!first) {
00278             sensor_list << "|";
00279         }
00280
00281         // Format: sensor_type:address (in hex)
00282         std::stringstream addr_hex;
00283         addr_hex << std::hex << static_cast<int>(sensor_info.second);
00284
00285         switch (sensor_info.first) {
00286             case SensorType::LIGHT:
00287                 sensor_list << "light:0x" << addr_hex.str();
00288                 break;
00289             case SensorType::ENVIRONMENT:
00290                 sensor_list << "environment:0x" << addr_hex.str();
00291                 break;
00292             default:
00293                 sensor_list << "unknown:0x" << addr_hex.str();
00294                 break;
00295         }
00296
00297         first = false;
00298     }
00299
00300     frames.push_back(frame_build(OperationType::VAL, SENSOR_GROUP, 2, sensor_list.str()));
00301     return frames;
00302 }

```

8.25 lib/comms/commands/storage_commands.cpp File Reference

```

#include "commands.h"
#include "communication.h"
#include "storage.h"
#include "filesystem/vfs.h"
#include "filesystem/littlefs.h"
#include <sys/stat.h>
#include <errno.h>
#include "dirent.h"

```

Macros

- #define STORAGE_GROUP 6
- #define LIST_FILES_COMMAND 0
- #define MOUNT_COMMAND 4

Functions

- `std::vector< Frame > handle_list_files` (const std::string ¶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
00014
00015 std::vector<Frame> handle\_list\_files([[maybe_unused]] const std::string& param, OperationType
operationType) {
00016     std::vector<Frame> frames;
00017     std::string error_msg;
00018
00019     if (operationType != OperationType::GET) {
00020         error_msg = error\_code\_to\_string(ErrorCode::INVALID\_OPERATION);
00021         frames.push_back(frame\_build(OperationType::ERR, STORAGE\_GROUP, LIST\_FILES\_COMMAND,
error_msg));
00022     }
00023     return frames;
00024 }
00025
00026 DIR* dir;
00027 struct dirent* ent;
00028 int file_count = 0;
```

```

00050     if ((dir = opendir("/")) != NULL) {
00051         // First, count the number of files
00052         while ((ent = readdir(dir)) != NULL) {
00053             const char* filename = ent->d_name;
00054             if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
00055                 continue;
00056             }
00057             file_count++;
00058         }
00059         closedir(dir);
00060
00061         // Send the number of files
00062         frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, LIST_FILES_COMMAND,
std::to_string(file_count)));
00063
00064         // Open the directory again to read file information
00065         dir = opendir("/");
00066         if (dir != NULL) {
00067             while ((ent = readdir(dir)) != NULL) {
00068                 const char* filename = ent->d_name;
00069
00070                 // Skip "." and ".." directories
00071                 if (strcmp(filename, ".") == 0 || strcmp(filename, "..") == 0) {
00072                     continue;
00073                 }
00074
00075                 // Get file size
00076                 char filepath[256];
00077                 int written = snprintf(filepath, sizeof(filepath), "%s", filename);
00078                 if (written < 0 || written >= static_cast<int>(sizeof(filepath))) {
00079                     continue; // Skip this file if path is too long
00080                 }
00081
00082                 FILE* file = fopen(filepath, "rb");
00083                 size_t file_size = 0;
00084
00085                 if (file != NULL) {
00086                     fseek(file, 0, SEEK_END);
00087                     file_size = ftell(file);
00088                     fclose(file);
00089                 }
00090
00091                 // Create and send frame with filename and size
00092                 char file_info[512];
00093                 snprintf(file_info, sizeof(file_info), "%s:%zu", filename, file_size);
00094                 frames.push_back(frame_build(OperationType::SEQ, STORAGE_GROUP, LIST_FILES_COMMAND,
file_info));
00095             }
00096             closedir(dir);
00097             frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, LIST_FILES_COMMAND,
"SEQ_DONE"));
00098             return frames;
00099         } else {
00100             error_msg = error_code_to_string(ErrorCode::INTERNAL_FAIL_TO_READ);
00101             frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND,
error_msg));
00102             return frames;
00103         }
00104     } else {
00105         frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, LIST_FILES_COMMAND,
error_msg));
00106         return frames;
00107     }
00108 }
00109
00126 std::vector<Frame> handle_mount(const std::string& param, OperationType operationType) {
00127     std::vector<Frame> frames;
00128     std::string error_msg;
00129
00130     if (operationType == OperationType::GET) {
00131         bool state = SystemStateManager::get_instance().is_sd_card_mounted();
00132
00133         frames.push_back(frame_build(OperationType::VAL, STORAGE_GROUP, MOUNT_COMMAND,
std::to_string(state)));
00134         return frames;
00135     } else if (operationType == OperationType::SET) {
00136         if (param == "1") {
00137             if (fs_init()) {
00138                 frames.push_back(frame_build(OperationType::RES, STORAGE_GROUP, MOUNT_COMMAND,
"SD_MOUNT_OK"));
00139                 return frames;
00140             } else {
00141                 error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
00142                 frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND,
error_msg));
00143                 return frames;
00144             }

```

```

00145         } else if (param == "0") {
00146             if (fs_unmount("/") == 0) {
00147                 if (SystemStateManager::get_instance().is_sd_card_mounted()) {
00148                     frames.push_back(frame_build(OperationType::RES, STORAGE_GROUP, MOUNT_COMMAND,
00149 "SD_UNMOUNT_OK"));
00149                 }
00150                 return frames;
00151             } else {
00152                 error_msg = error_code_to_string(ErrorCode::FAIL_TO_SET);
00153                 frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND,
00154 error_msg));
00154                 return frames;
00155             }
00156         } else {
00157             error_msg = error_code_to_string(ErrorCode::PARAM_INVALID);
00158             frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND,
00159 error_msg));
00159             return frames;
00160         }
00161     } else {
00162         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00163         frames.push_back(frame_build(OperationType::ERR, STORAGE_GROUP, MOUNT_COMMAND, error_msg));
00164         return frames;
00165     }
00166 } // StorageCommands

```

8.27 lib/comms/commands/telemetry_commands.cpp File Reference

```

#include "commands.h"
#include "communication.h"
#include "telemetry_manager.h"

```

Macros

- `#define` [TELEMETRY_GROUP](#) 8
- `#define` [GET_LAST_TELEMETRY_RECORD_COMMAND](#) 2
- `#define` [GET_LAST_SENSOR_RECORD_COMMAND](#) 3

Functions

- `std::vector< Frame >` [handle_get_last_telemetry_record](#) (const std::string ¶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.

8.27.1 Macro Definition Documentation

8.27.1.1 TELEMETRY_GROUP

```
#define TELEMETRY_GROUP 8
```

Definition at line 5 of file [telemetry_commands.cpp](#).

8.27.1.2 GET_LAST_TELEMETRY_RECORD_COMMAND

```
#define GET_LAST_TELEMETRY_RECORD_COMMAND 2
```

Definition at line 6 of file [telemetry_commands.cpp](#).

8.27.1.3 GET_LAST_SENSOR_RECORD_COMMAND

```
#define GET_LAST_SENSOR_RECORD_COMMAND 3
```

Definition at line 7 of file [telemetry_commands.cpp](#).

8.28 telemetry_commands.cpp

[Go to the documentation of this file.](#)

```
00001 #include "commands.h"
00002 #include "communication.h"
00003 #include "telemetry_manager.h"
00004
00005 #define TELEMETRY_GROUP 8
00006 #define GET_LAST_TELEMETRY_RECORD_COMMAND 2
00007 #define GET_LAST_SENSOR_RECORD_COMMAND 3
00008
00014
00032 std::vector<Frame> handle_get_last_telemetry_record([[maybe_unused]] const std::string& param,
OperationType operationType) {
00033     std::vector<Frame> frames;
00034     std::string error_msg;
00035
00036     if (operationType != OperationType::GET) {
00037         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00038         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP,
GET_LAST_TELEMETRY_RECORD_COMMAND, error_msg));
00039         return frames;
00040     }
00041
00042     std::string csv_data = TelemetryManager::get_instance().get_last_telemetry_record_csv();
00043
00044     if (csv_data.empty()) {
00045         error_msg = "NO_DATA";
00046         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP,
GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00047         return frames;
00048     }
00049     frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP,
GET_LAST_TELEMETRY_RECORD_COMMAND, csv_data));
00050
00051     return frames;
00052 }
00053
00054
00065 std::vector<Frame> handle_get_last_sensor_record([[maybe_unused]] const std::string& param,
OperationType operationType) {
00066     std::vector<Frame> frames;
00067     std::string error_msg;
00068
00069     if (operationType != OperationType::GET) {
00070         error_msg = error_code_to_string(ErrorCode::INVALID_OPERATION);
00071         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP,
GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00072         return frames;
00073     }
00074
00075     std::string csv_data = TelemetryManager::get_instance().get_last_sensor_record_csv();
00076
00077     if (csv_data.empty()) {
00078         error_msg = "NO_DATA";
00079         frames.push_back(frame_build(OperationType::ERR, TELEMETRY_GROUP,
GET_LAST_SENSOR_RECORD_COMMAND, error_msg));
00080         return frames;
00081     }
00082     frames.push_back(frame_build(OperationType::VAL, TELEMETRY_GROUP, GET_LAST_SENSOR_RECORD_COMMAND,
csv_data));
00083
00084     return frames;
00085 }
00086 } // TelemetryBufferCommands
```

8.29 lib/comms/communication.cpp File Reference

```
#include "communication.h"
```

Functions

- bool [initialize_radio](#) ()
Initializes the LoRa radio module.
- void [lora_tx_done_callback](#) ()
Callback function for LoRa transmission completion.

Variables

- string [outgoing](#)
- uint8_t [msgCount](#) = 0
- long [lastSendTime](#) = 0
- long [lastReceiveTime](#) = 0
- long [lastPrintTime](#) = 0
- unsigned long [interval](#) = 0

8.29.1 Function Documentation

8.29.1.1 initialize_radio()

```
bool initialize_radio ()
```

Initializes the LoRa radio module.

Returns

True if initialization was successful, false otherwise.

Sets the LoRa pins and attempts to begin LoRa communication at a specified frequency. Emits a [CommsEvent::RADIO_INIT](#) event on success or a [CommsEvent::RADIO_ERROR](#) event on failure.

Definition at line 16 of file [communication.cpp](#).

8.29.1.2 lora_tx_done_callback()

```
void lora_tx_done_callback ()
```

Callback function for LoRa transmission completion.

Prints a debug message to the UART and sets the LoRa module to receive mode.

Definition at line 44 of file [communication.cpp](#).

8.29.2 Variable Documentation

8.29.2.1 outgoing

```
string outgoing
```

Definition at line 3 of file [communication.cpp](#).

8.29.2.2 msgCount

```
uint8_t msgCount = 0
```

Definition at line 4 of file [communication.cpp](#).

8.29.2.3 lastSendTime

```
long lastSendTime = 0
```

Definition at line 5 of file [communication.cpp](#).

8.29.2.4 lastReceiveTime

```
long lastReceiveTime = 0
```

Definition at line 6 of file [communication.cpp](#).

8.29.2.5 lastPrintTime

```
long lastPrintTime = 0
```

Definition at line 7 of file [communication.cpp](#).

8.29.2.6 interval

```
unsigned long interval = 0
```

Definition at line 8 of file [communication.cpp](#).

8.30 communication.cpp

[Go to the documentation of this file.](#)

```

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
00016 bool initialize_radio() {
00017     LoRa.set_pins(lora_cs_pin, lora_reset_pin, lora_irq_pin);
00018     long frequency = 433E6;
00019     bool init_status = false;
00020     if (!LoRa.begin(frequency))
00021     {
00022         uart_print("LoRa init failed. Check your connections.", VerbosityLevel::WARNING);
00023         init_status = false;
00024     } else {
00025         uart_print("LoRa initialized with frequency " + std::to_string(frequency),
VerbosityLevel::INFO);
00026
00027         // Set up TxDone callback to automatically return to receive mode
00028         LoRa.onTxDone(lora_tx_done_callback);
00029
00030         LoRa.receive(0);
00031
00032         init_status = true;
00033     }
00034
00035     EventEmitter::emit(EventGroup::COMMS, init_status ? CommsEvent::RADIO_INIT :
CommsEvent::RADIO_ERROR);
00036
00037     return init_status;
00038 }
00039
00044 void lora_tx_done_callback() {
00045     uart_print("LoRa transmission complete", VerbosityLevel::DEBUG);
00046     LoRa.receive(0);
00047 }

```

8.31 lib/comms/communication.h File Reference

```

#include <string>
#include <vector>
#include "protocol.h"
#include "event_manager.h"

```

Functions

- bool [initialize_radio](#) ()
Initializes the LoRa radio module.
- void [lora_tx_done_callback](#) ()
Callback function for LoRa transmission completion.
- void [on_receive](#) (int packetSize)
Callback function for handling received LoRa packets.
- void [handle_uart_input](#) ()
Handles UART input.
- void [send_message](#) (std::string outgoing)
- void [send_frame_uart](#) (const [Frame](#) &frame)
- void [send_frame_lora](#) (const [Frame](#) &frame)
- void [split_and_send_message](#) (const uint8_t *data, size_t length)

- `std::vector< Frame > execute_command` (`uint32_t commandKey`, `const std::string ¶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 16 of file [communication.cpp](#).

8.31.1.2 lora_tx_done_callback()

```
void lora_tx_done_callback ()
```

Callback function for LoRa transmission completion.

Prints a debug message to the UART and sets the LoRa module to receive mode.

Definition at line 44 of file [communication.cpp](#).

8.31.1.3 send_message()

```
void send_message (
    std::string outgoing)
```

8.31.1.4 send_frame_uart()

```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 47 of file [send.cpp](#).

8.31.1.5 send_frame_lora()

```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 39 of file [send.cpp](#).

8.31.1.6 split_and_send_message()

```
void split_and_send_message (
    const uint8_t * data,
    size_t length)
```

8.31.1.7 determine_unit()

```
std::string determine_unit (
    uint8_t group,
    uint8_t command)
```

8.32 communication.h

[Go to the documentation of this file.](#)

```
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
operationType);
00020
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,
const ValueUnit unitType = ValueUnit::UNDEFINED);
00025
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"
```

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
00014
00037 std::string frame_encode(const Frame& frame) {
00038     std::stringstream ss;
00039     ss << static_cast<int>(frame.direction) << DELIMITER
00040         << operation_type_to_string(frame.operationType) << DELIMITER
00041         << static_cast<int>(frame.group) << DELIMITER
00042         << static_cast<int>(frame.command) << DELIMITER
00043         << frame.value;
00044
00045     if (!frame.unit.empty()) {
00046         ss << DELIMITER << frame.unit;
00047     }
00048
00049     return FRAME_BEGIN + DELIMITER + ss.str() + DELIMITER + FRAME_END;
00050 }
00051
00052
00062 Frame frame_decode(const std::string& data) {
00063     try {
00064         Frame frame;
00065         std::stringstream ss(data);
00066         std::string token;
00067
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
00117 void frame_process(const std::string& data, Interface interface) {
00118     gpio_put(PICO_DEFAULT_LED_PIN, 0);
00119
00120     uart_print("Processing frame: " + data, VerbosityLevel::INFO);
00121     try {
00122         Frame frame = frame_decode(data);
00123         uint32_t command_key = (static_cast<uint32_t>(frame.group) << 8) |
                                static_cast<uint32_t>(frame.command);
00124

```

```

00125         std::vector<Frame> response_frames = execute_command(command_key, frame.value,
00126 frame.operationType);
00127         gpio_put(PICO_DEFAULT_LED_PIN, 1);
00128
00129         // Send all responses through the same interface that received the command
00130         for (const auto& response_frame : response_frames) {
00131             if (interface == Interface::UART) {
00132                 send_frame_uart(response_frame);
00133             } else if (interface == Interface::LORA) {
00134                 send_frame_lora(response_frame);
00135                 sleep_ms(25);
00136             }
00137         }
00138     } catch (const std::exception& e) {
00139         Frame error_frame = frame_build(OperationType::ERR, 0, 0, e.what());
00140         if (interface == Interface::UART) {
00141             send_frame_uart(error_frame);
00142         } else if (interface == Interface::LORA) {
00143             send_frame_lora(error_frame);
00144         }
00145     }
00146 }
00147
00158 Frame frame_build(OperationType operation, uint8_t group, uint8_t command,
00159 const std::string& value, const ValueUnit unitType) {
00160     Frame frame;
00161     frame.header = FRAME_BEGIN;
00162     frame.footer = FRAME_END;
00163
00164     switch (operation) {
00165         case OperationType::VAL:
00166             frame.direction = 1;
00167             frame.operationType = OperationType::VAL;
00168             frame.value = value;
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         default:
00193             break;
00194     }
00195
00196     frame.group = group;
00197     frame.command = command;
00198
00199     return frame;
00200 }
00201 // 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 <stdio>
#include <stdlib>
#include <string>
#include "utils.h"
#include "time.h"
#include "build_number.h"
#include "LoRa/LoRa-RP2040.h"
```

Classes

- struct [Frame](#)
Represents a communication frame used for data exchange.

Enumerations

- enum class [ErrorCode](#) {
[ErrorCode::PARAM_UNNECESSARY](#) , [ErrorCode::PARAM_REQUIRED](#) , [ErrorCode::PARAM_INVALID](#) ,
[ErrorCode::INVALID_OPERATION](#) ,
[ErrorCode::NOT_ALLOWED](#) , [ErrorCode::INVALID_FORMAT](#) , [ErrorCode::INVALID_VALUE](#) , [ErrorCode::FAIL_TO_SET](#)
, [ErrorCode::INTERNAL_FAIL_TO_READ](#) , [ErrorCode::UNKNOWN_ERROR](#) }
Standard error codes for command responses.
- enum class [OperationType](#) {
[OperationType::GET](#) , [OperationType::SET](#) , [OperationType::RES](#) , [OperationType::VAL](#) ,
[OperationType::SEQ](#) , [OperationType::ERR](#) }
Represents the type of operation being performed.
- enum class [CommandAccessLevel](#) { [CommandAccessLevel::NONE](#) , [CommandAccessLevel::READ_ONLY](#)
, [CommandAccessLevel::WRITE_ONLY](#) , [CommandAccessLevel::READ_WRITE](#) }
Represents the access level required to execute a command.
- enum class [ValueUnit](#) {
[ValueUnit::UNDEFINED](#) , [ValueUnit::SECOND](#) , [ValueUnit::VOLT](#) , [ValueUnit::BOOL](#) ,
[ValueUnit::DATETIME](#) , [ValueUnit::TEXT](#) , [ValueUnit::MILIAMP](#) }
Represents the unit of measurement for a payload value.
- enum class [ExceptionType](#) {
[ExceptionType::NONE](#) , [ExceptionType::NOT_ALLOWED](#) , [ExceptionType::INVALID_PARAM](#) , [ExceptionType::INVALID_OPER](#)
, [ExceptionType::PARAM_UNNECESSARY](#) }
Represents the type of exception that occurred during command execution.
- enum class [Interface](#) { [Interface::UART](#) , [Interface::LORA](#) }
Represents the communication interface being used.

Functions

- std::string [exception_type_to_string](#) ([ExceptionType](#) type)
Converts an [ExceptionType](#) to a string.
- std::string [error_code_to_string](#) ([ErrorCode](#) code)
Converts an [ErrorCode](#) to its string representation.
- std::string [operation_type_to_string](#) ([OperationType](#) type)
Converts an [OperationType](#) to a string.
- [OperationType string_to_operation_type](#) (const std::string &str)

Converts a string to an [OperationType](#).

- `std::vector< uint8_t > hex_string_to_bytes (const std::string &hexString)`

Converts a hex string to a vector of bytes.

- `std::string value_unit_type_to_string (ValueUnit unit)`

Converts a [ValueUnit](#) to a string.

Variables

- `const std::string FRAME_BEGIN = "KBST"`
- `const std::string FRAME_END = "TSBK"`
- `const char DELIMITER = ';' ;`

8.35.1 Variable Documentation

8.35.1.1 [FRAME_BEGIN](#)

```
const std::string FRAME_BEGIN = "KBST"
```

Definition at line 31 of file [protocol.h](#).

8.35.1.2 [FRAME_END](#)

```
const std::string FRAME_END = "TSBK"
```

Definition at line 38 of file [protocol.h](#).

8.35.1.3 [DELIMITER](#)

```
const char DELIMITER = ';' ;
```

Definition at line 45 of file [protocol.h](#).

8.36 protocol.h

[Go to the documentation of this file.](#)

```
00001 // protocol.h
00002 #ifndef PROTOCOL_H
00003 #define PROTOCOL_H
00004
00005 #include <string>
00006 #include <map>
00007 #include <functional>
00008 #include <vector>
00009 #include <stdint>
00010 #include <iomanip>
00011 #include "pin_config.h"
00012 #include "PowerManager.h"
00013 #include <cstdio>
00014 #include <cstdlib>
00015 #include <map>
00016 #include <cstring>
00017 #include "utils.h"
00018 #include "time.h"
00019 #include "build_number.h"
```

```

00020 #include "LoRa/LoRa-RP2040.h"
00021
00031 const std::string FRAME_BEGIN = "KBST";
00032
00038 const std::string FRAME_END = "TSBK";
00039
00045 const char DELIMITER = ',';
00046
00047
00053 enum class ErrorCode {
00054     PARAM_UNNECESSARY,    // Parameter provided but not needed
00055     PARAM_REQUIRED,       // Required parameter missing
00056     PARAM_INVALID,        // Parameter has invalid format or value
00057     INVALID_OPERATION,    // Operation not allowed for this command
00058     NOT_ALLOWED,          // Operation not permitted
00059     INVALID_FORMAT,       // Input format is incorrect
00060     INVALID_VALUE,        // Value is outside expected range
00061     FAIL_TO_SET,          // Failed to set provided value
00062     INTERNAL_FAIL_TO_READ, // Failed to read from device in remote
00063     UNKNOWN_ERROR        // Generic error
00064 };
00065
00066
00072 enum class OperationType {
00074     GET,
00076     SET,
00078     RES,
00080     VAL,
00082     SEQ,
00084     ERR,
00085 };
00086
00087
00088
00089
00095 enum class CommandAccessLevel {
00097     NONE,
00099     READ_ONLY,
00101     WRITE_ONLY,
00103     READ_WRITE
00104 };
00105
00106
00107
00113 enum class ValueUnit {
00115     UNDEFINED,
00117     SECOND,
00119     VOLT,
00121     BOOL,
00123     DATETIME,
00125     TEXT,
00127     MILLIAMP,
00128 };
00129
00130
00131
00137 enum class ExceptionType {
00139     NONE,
00141     NOT_ALLOWED,
00143     INVALID_PARAM,
00145     INVALID_OPERATION,
00147     PARAM_UNNECESSARY
00148 };
00149
00150
00151
00157 enum class Interface {
00159     UART,
00161     LORA
00162 };
00163
00164
00181 struct Frame {
00182     std::string header;           // Start marker
00183     uint8_t direction;           // 0 = ground->sat, 1 = sat->ground
00184     OperationType operationType;
00185     uint8_t group;               // Group ID
00186     uint8_t command;            // Command ID within group
00187     std::string value;           // Payload value
00188     std::string unit;            // Payload unit
00189     std::string footer;          // End marker
00190 };
00191
00192 std::string exception_type_to_string(ExceptionType type);
00193 std::string error_code_to_string(ErrorCode code);
00194 std::string operation_type_to_string(OperationType type);
00195 OperationType string_to_operation_type(const std::string& str);

```



```
00196 std::vector<uint8_t> hex_string_to_bytes(const std::string& hexString);
00197 std::string value_unit_type_to_string(ValueUnit unit);
00198
00199 #endif
00200
```

8.37 lib/comms/receive.cpp File Reference

Implements functions for receiving and processing data, including LoRa and UART input.

```
#include "communication.h"
```

Macros

- `#define MAX_PACKET_SIZE 255`

Functions

- void `on_receive` (int packet_size)
Callback function for handling received LoRa packets.
- void `handle_uart_input` ()
Handles UART input.

8.37.1 Detailed Description

Implements functions for receiving and processing data, including LoRa and UART input.

Definition in file [receive.cpp](#).

8.37.2 Macro Definition Documentation

8.37.2.1 MAX_PACKET_SIZE

```
#define MAX_PACKET_SIZE 255
```

Definition at line 3 of file [receive.cpp](#).

8.38 receive.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003 #define MAX_PACKET_SIZE 255
00004
00011
00019 void on_receive(int packet_size) {
00020     if (packet_size == 0) return;
00021     uart_print("Received LoRa packet of size " + std::to_string(packet_size), VerbosityLevel::DEBUG);
00022
00023     std::vector<uint8_t> buffer;
00024     buffer.reserve(packet_size);
00025
00026     int bytes_read = 0;
00027
00028     while (LoRa.available() && bytes_read < packet_size) {
00029         if (bytes_read >= MAX_PACKET_SIZE) {
00030             uart_print("Error: Packet exceeds maximum allowed size!", VerbosityLevel::ERROR);
00031             return;
00032         }
00033         buffer.push_back(LoRa.read());
00034         bytes_read++;
00035     }
00036
00037     if (bytes_read < 2) {
00038         uart_print("Error: Packet too small to contain metadata!", VerbosityLevel::ERROR);
00039         return;
00040     }
00041
00042     uart_print("Received " + std::to_string(bytes_read) + " bytes", VerbosityLevel::DEBUG);
00043
00044     uint8_t received_destination = buffer[0];
00045     uint8_t received_local_address = buffer[1];
00046
00047     if (received_destination != lora_address_local) {
00048         uart_print("Error: Destination address mismatch!", VerbosityLevel::ERROR);
00049         return;
00050     }
00051
00052     if (received_local_address != lora_address_remote) {
00053         uart_print("Error: Local address mismatch!", VerbosityLevel::ERROR);
00054         return;
00055     }
00056
00057     // Skip 2 bytes being local and remote address appended by ground station
00058     int start_index = 2;
00059     std::string received(buffer.begin() + start_index, buffer.end());
00060
00061     if (received.empty()) return;
00062
00063     std::stringstream hex_dump;
00064     hex_dump << "Raw bytes: ";
00065     for (int i = 0; i < bytes_read; i++) {
00066         hex_dump << std::hex << std::setfill('0') << std::setw(2)
00067             << static_cast<int>(buffer[i]) << " ";
00068     }
00069     uart_print(hex_dump.str(), VerbosityLevel::DEBUG);
00070
00071     size_t header_pos = received.find(FRAME_BEGIN);
00072     size_t footer_pos = received.find(FRAME_END);
00073
00074     if (header_pos != std::string::npos && footer_pos != std::string::npos && footer_pos > header_pos)
00075     {
00076         std::string frame_data = received.substr(header_pos, footer_pos + FRAME_END.length() -
00077         header_pos);
00078         uart_print("Extracted frame (length=" + std::to_string(frame_data.length()) + "): " +
00079         frame_data, VerbosityLevel::DEBUG);
00080         frame_process(frame_data, Interface::LORA);
00081     }
00082     else {
00083         uart_print("No valid frame found in received data", VerbosityLevel::WARNING);
00084     }
00085 }
00086
00087
00088
00089
00090 void handle_uart_input() {
00091     static std::string uart_buffer;
00092
00093     while (uart_is_readable(DEBUG_UART_PORT)) {
00094         char c = uart_getc(DEBUG_UART_PORT);
00095
00096         if (c == '\r' || c == '\n') {
00097             uart_print("Received UART string: " + uart_buffer, VerbosityLevel::DEBUG);
00098             frame_process(uart_buffer, Interface::UART);
00099         }
00100         uart_buffer += c;
00101     }
00102 }

```

```

00099         uart_buffer.clear();
00100     } else {
00101         uart_buffer += c;
00102     }
00103 }
00104 }

```

8.39 lib/comms/send.cpp File Reference

Implements functions for sending data, including LoRa messages and Frames.

```
#include "communication.h"
```

Functions

- void [send_message](#) (string [outgoing](#))
Sends a message using LoRa.
- void [send_frame_lora](#) (const [Frame](#) &frame)
- void [send_frame_uart](#) (const [Frame](#) &frame)

8.39.1 Detailed Description

Implements functions for sending data, including LoRa messages and Frames.

Definition in file [send.cpp](#).

8.39.2 Function Documentation

8.39.2.1 [send_message\(\)](#)

```
void send_message (
    string outgoing)
```

Sends a message using LoRa.

Parameters

<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](#).

8.39.2.2 [send_frame_lora\(\)](#)

```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 39 of file [send.cpp](#).

8.39.2.3 send_frame_uart()

```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 47 of file [send.cpp](#).

8.40 send.cpp

[Go to the documentation of this file.](#)

```
00001 #include "communication.h"
00002
00003
00004
00015 void send_message(string outgoing)
00016 {
00017     std::vector<char> send(outgoing.length() + 1);
00018     strcpy(send.data(), outgoing.c_str());
00019
00020     uart_print("LoRa packet begin", VerbosityLevel::DEBUG);
00021     LoRa.beginPacket(); // start packet
00022     LoRa.write(lora_address_remote); // add destination address
00023     LoRa.write(lora_address_local); // add sender address
00024     LoRa.print(send.data()); // add payload
00025     LoRa.endPacket(false); // finish packet and send it, param - async
00026
00027     uart_print("LoRa packet end", VerbosityLevel::DEBUG);
00028
00029     std::string message_to_log = "Sent message of size " + std::to_string(send.size());
00030     message_to_log += " to 0x" + std::to_string(lora_address_remote);
00031     message_to_log += " containing: " + string(send.data());
00032
00033     uart_print(message_to_log, VerbosityLevel::DEBUG);
00034
00035     LoRa.flush();
00036 }
00037
00038
00039 void send_frame_lora(const Frame& frame) {
00040     uart_print("Sending frame via LoRa", VerbosityLevel::DEBUG);
00041     string outgoing = frame_encode(frame);
00042     send_message(outgoing);
00043     uart_print("Frame sent via LoRa", VerbosityLevel::DEBUG);
00044 }
00045
00046 // If level is 0 - SILENT it means no diagnostic output but frame communications should still work
00047 void send_frame_uart(const Frame& frame) {
00048     std::string encoded_frame = frame_encode(frame);
00049     uart_print(encoded_frame, VerbosityLevel::SILENT);
00050 }
00051
00052
```

8.41 lib/comms/utils_converters.cpp File Reference

Implements utility functions for converting between different data types.

```
#include "communication.h"
```

Functions

- `std::string exception_type_to_string` (`ExceptionType` type)
Converts an `ExceptionType` to a string.
- `std::string value_unit_type_to_string` (`ValueUnit` unit)
Converts a `ValueUnit` to a string.
- `std::string operation_type_to_string` (`OperationType` type)
Converts an `OperationType` to a string.
- `OperationType string_to_operation_type` (const `std::string` &str)
Converts a string to an `OperationType`.
- `std::string error_code_to_string` (`ErrorCode` code)
Converts an `ErrorCode` to its string representation.
- `std::vector< uint8_t > hex_string_to_bytes` (const `std::string` &hexString)
Converts a hex string to a vector of bytes.

8.41.1 Detailed Description

Implements utility functions for converting between different data types.

Definition in file [utils_converters.cpp](#).

8.42 utils_converters.cpp

[Go to the documentation of this file.](#)

```

00001 #include "communication.h"
00002
00003
00009
00016 std::string exception_type_to_string(ExceptionType type) {
00017     switch (type) {
00018         case ExceptionType::NOT_ALLOWED:      return "NOT ALLOWED";
00019         case ExceptionType::INVALID_PARAM:     return "INVALID PARAM";
00020         case ExceptionType::INVALID_OPERATION: return "INVALID OPERATION";
00021         case ExceptionType::PARAM_UNECESSARY:  return "PARAM UNECESSARY";
00022         case ExceptionType::NONE:              return "NONE";
00023         default:                               return "UNKNOWN EXCEPTION";
00024     }
00025 }
00026
00027
00034 std::string value_unit_type_to_string(ValueUnit unit) {
00035     switch (unit) {
00036         case ValueUnit::UNDEFINED: return "";
00037         case ValueUnit::SECOND:    return "s";
00038         case ValueUnit::VOLT:      return "V";
00039         case ValueUnit::BOOL:      return "";
00040         case ValueUnit::DATETIME:  return "";
00041         case ValueUnit::TEXT:      return "";
00042         case ValueUnit::MILIAMP:   return "mA";
00043         default: return "";
00044     }
00045 }
00046
00047
00054 std::string operation_type_to_string(OperationType type) {
00055     switch (type) {
00056         case OperationType::GET: return "GET";
00057         case OperationType::SET: return "SET";
00058         case OperationType::VAL: return "VAL";
00059         case OperationType::ERR: return "ERR";
00060         case OperationType::RES: return "RES";
00061         case OperationType::SEQ: return "SEQ";
00062         default: return "UNKNOWN";
00063     }
00064 }
00065

```

```

00066
00073 OperationType string_to_operation_type(const std::string& str) {
00074     if (str == "GET") return OperationType::GET;
00075     if (str == "SET") return OperationType::SET;
00076     if (str == "VAL") return OperationType::VAL;
00077     if (str == "ERR") return OperationType::ERR;
00078     if (str == "RES") return OperationType::RES;
00079     if (str == "SEQ") return OperationType::SEQ;
00080     return OperationType::GET; // Default to GET
00081 }
00082
00089 std::string error_code_to_string(ErrorCode code) {
00090     switch (code) {
00091         case ErrorCode::PARAM_UNNECESSARY: return "PARAM_UNNECESSARY";
00092         case ErrorCode::PARAM_REQUIRED: return "PARAM_REQUIRED";
00093         case ErrorCode::PARAM_INVALID: return "PARAM_INVALID";
00094         case ErrorCode::INVALID_OPERATION: return "INVALID_OPERATION";
00095         case ErrorCode::NOT_ALLOWED: return "NOT_ALLOWED";
00096         case ErrorCode::INVALID_FORMAT: return "INVALID_FORMAT";
00097         case ErrorCode::INVALID_VALUE: return "INVALID_VALUE";
00098         case ErrorCode::FAIL_TO_SET: return "FAIL_TO_SET";
00099         case ErrorCode::INTERNAL_FAIL_TO_READ: return "INTERNAL_FAIL_TO_READ";
00100         default: return "UNKNOWN_ERROR";
00101     }
00102 }
00103
00104
00111 std::vector<uint8_t> hex_string_to_bytes(const std::string& hexString) {
00112     std::vector<uint8_t> bytes;
00113     for (size_t i = 0; i < hexString.length(); i += 2) {
00114         std::string byteString = hexString.substr(i, 2);
00115         unsigned int byte;
00116         std::stringstream ss;
00117         ss << std::hex << byteString;
00118         ss >> byte;
00119         bytes.push_back(static_cast<uint8_t>(byte));
00120     }
00121     return bytes;
00122 }
00123

```

8.43 lib/eventman/event_manager.cpp File Reference

Implementation of the Event Manager and Event Emitter classes.

```

#include "event_manager.h"
#include <cstdio>
#include "protocol.h"
#include "pico/multicore.h"
#include "communication.h"
#include "utils.h"
#include "DS3231.h"

```

8.43.1 Detailed Description

Implementation of the Event Manager and Event Emitter classes.

This file implements the [EventManager](#) class, which provides a singleton instance for logging events to a circular buffer and saving them to persistent storage. The [EventEmitter](#) class provides a simple interface for emitting events throughout the system.

Definition in file [event_manager.cpp](#).

8.44 event_manager.cpp

[Go to the documentation of this file.](#)

```

00001
00015
00016 #include "event_manager.h"
00017 #include <cstdio>
00018 #include "protocol.h"
00019 #include "pico/multicore.h"
00020 #include "communication.h"
00021 #include "utils.h"
00022 #include "DS3231.h"
00023
00024
00030 bool EventManager::init() {
00031     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00032         uart_print("Event manager initialized (storage not available)", VerbosityLevel::WARNING);
00033         return false;
00034     }
00035
00036     FILE* file = fopen(EVENT_LOG_FILE, "w");
00037     if (!file) {
00038         file = fopen(EVENT_LOG_FILE, "w");
00039         if (file) {
00040             fclose(file);
00041             uart_print("Created new event log", VerbosityLevel::INFO);
00042         }
00043         else {
00044             uart_print("Failed to create event log", VerbosityLevel::ERROR);
00045             return false;
00046         }
00047     }
00048     else {
00049         fclose(file);
00050     }
00051
00052     uart_print("Event manager initialized", VerbosityLevel::INFO);
00053     return true;
00054 }
00055
00056
00063 void EventManager::log_event(uint8_t group, uint8_t event) {
00064     mutex_enter_blocking(&eventMutex);
00065
00066     uint32_t timestamp = DS3231::get_instance().get_local_time();
00067     uint16_t id = nextEventId++;
00068
00069     EventLog& log = events[writeIndex];
00070     log.id = id;
00071     log.timestamp = timestamp;
00072     log.group = group;
00073     log.event = event;
00074
00075     writeIndex = (writeIndex + 1) % EVENT_BUFFER_SIZE;
00076     if (eventCount < EVENT_BUFFER_SIZE) {
00077         eventCount++;
00078     }
00079
00080     eventsSinceFlush++;
00081
00082     mutex_exit(&eventMutex);
00083
00084     std::string event_string = "Event: " + std::to_string(id) +
00085         " Group: " + std::to_string(group) +
00086         " Event: " + std::to_string(event);
00087     uart_print(event_string, VerbosityLevel::WARNING);
00088
00089     if (eventsSinceFlush >= EVENT_FLUSH_THRESHOLD || group == static_cast<uint8_t>(EventGroup::POWER))
00090     {
00091         save_to_storage();
00092         eventsSinceFlush = 0;
00093     }
00094
00095
00102 const EventLog& EventManager::get_event(size_t index) const {
00103     mutex_enter_blocking(const_cast<mutex_t*>(&eventMutex));
00104     if (index >= eventCount) {
00105         static EventLog emptyEvent;
00106         mutex_exit(const_cast<mutex_t*>(&eventMutex));
00107         return emptyEvent;
00108     }
00109
00110     size_t readIndex;
00111     if (eventCount == EVENT_BUFFER_SIZE) {

```

```

00112         readIndex = (writeIndex + index) % EVENT_BUFFER_SIZE;
00113     }
00114     else {
00115         readIndex = index;
00116     }
00117     const EventLog& event = events[readIndex];
00118     mutex_exit(const_cast<mutex_t*>(&eventMutex));
00119     return event;
00120 }
00121
00122
00128 bool EventManager::save_to_storage() {
00129     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00130         bool status = fs_init();
00131         if (!status) {
00132             return false;
00133         }
00134     }
00135
00136     FILE* file = fopen(EVENT_LOG_FILE, "a");
00137     if (file) {
00138         size_t startIdx = (writeIndex >= eventsSinceFlush) ?
00139             writeIndex - eventsSinceFlush :
00140             EVENT_BUFFER_SIZE - (eventsSinceFlush - writeIndex);
00141
00142         for (size_t i = 0; i < eventsSinceFlush; i++) {
00143             size_t idx = (startIdx + i) % EVENT_BUFFER_SIZE;
00144             fprintf(file, "%u;%lu;%u;%u\n",
00145                 events[idx].id,
00146                 events[idx].timestamp,
00147                 events[idx].group,
00148                 events[idx].event
00149             );
00150         }
00151         fclose(file);
00152         uart_print("Events saved to storage", VerbosityLevel::INFO);
00153         return true;
00154     }
00155     return false;
00156 }
00157

```

8.45 lib/eventman/event_manager.h File Reference

Header file for the Event Manager and Event Emitter classes.

```

#include "PowerManager.h"
#include <stdint>
#include <string>
#include "pico/mutex.h"
#include "storage.h"
#include "utils.h"
#include "system_state_manager.h"

```

Classes

- class [EventLog](#)
Structure for storing event log data.
- class [EventManager](#)
Manages event logging and storage.
- class [EventEmitter](#)
Provides a simple interface for emitting events.

Macros

- #define `EVENT_BUFFER_SIZE` 100
Size of the event buffer.
- #define `EVENT_FLUSH_THRESHOLD` 10
Number of events to accumulate before flushing to storage.
- #define `EVENT_LOG_FILE` "/event_log.csv"
Path to the event log file.

Enumerations

- enum class `EventGroup` : `uint8_t` {
 `EventGroup::SYSTEM` = 0x00 , `EventGroup::POWER` = 0x01 , `EventGroup::COMMS` = 0x02 ,
 `EventGroup::GPS` = 0x03 ,
 `EventGroup::CLOCK` = 0x04 }
Enumeration of event groups.
- enum class `SystemEvent` : `uint8_t` {
 `SystemEvent::BOOT` = 0x01 , `SystemEvent::SHUTDOWN` = 0x02 , `SystemEvent::WATCHDOG_RESET` =
 0x03 , `SystemEvent::CORE1_START` = 0x04 ,
 `SystemEvent::CORE1_STOP` = 0x05 }
Enumeration of system events.
- enum class `PowerEvent` : `uint8_t` {
 `PowerEvent::BATTERY_LOW` = 0x01 , `PowerEvent::BATTERY_FULL` = 0x02 , `PowerEvent::POWER_FALLING`
 = 0x03 , `PowerEvent::BATTERY_NORMAL` = 0x04 ,
 `PowerEvent::SOLAR_ACTIVE` = 0x05 , `PowerEvent::SOLAR_INACTIVE` = 0x06 , `PowerEvent::USB_CONNECTED`
 = 0x07 , `PowerEvent::USB_DISCONNECTED` = 0x08 }
Enumeration of power events.
- enum class `CommsEvent` : `uint8_t` {
 `CommsEvent::RADIO_INIT` = 0x01 , `CommsEvent::RADIO_ERROR` = 0x02 , `CommsEvent::MSG_RECEIVED`
 = 0x03 , `CommsEvent::MSG_SENT` = 0x04 ,
 `CommsEvent::UART_ERROR` = 0x06 }
Enumeration of communications events.
- enum class `GPSEvent` : `uint8_t` {
 `GPSEvent::LOCK` = 0x01 , `GPSEvent::LOST` = 0x02 , `GPSEvent::ERROR` = 0x03 , `GPSEvent::POWER_ON`
 = 0x04 ,
 `GPSEvent::POWER_OFF` = 0x05 , `GPSEvent::DATA_READY` = 0x06 , `GPSEvent::PASS_THROUGH_START`
 = 0x07 , `GPSEvent::PASS_THROUGH_END` = 0x08 }
Enumeration of GPS events.
- enum class `ClockEvent` : `uint8_t` { `ClockEvent::CHANGED` = 0x01 , `ClockEvent::GPS_SYNC` = 0x02 ,
 `ClockEvent::GPS_SYNC_DATA_NOT_READY` = 0x03 }
Enumeration of clock events.

Functions

- class `EventLog __attribute__((packed))`

Variables

- `uint16_t id`
Unique identifier for the event.
- `uint32_t timestamp`
Timestamp of the event in milliseconds since boot.
- `uint8_t group`
Event group.
- `uint8_t event`
Event code.
- class `EventManager __attribute__((packed))`

8.45.1 Detailed Description

Header file for the Event Manager and Event Emitter classes.

This file defines the classes and enumerations necessary for managing and emitting system events. The [EventManager](#) class provides a singleton instance for logging events to a circular buffer and saving them to persistent storage. The [EventEmitter](#) class provides a simple interface for emitting events throughout the system.

Definition in file [event_manager.h](#).

8.45.2 Variable Documentation

8.45.2.1 id

```
uint16_t id
```

Unique identifier for the event.

Definition at line 2 of file [event_manager.h](#).

8.45.2.2 timestamp

```
uint32_t timestamp
```

Timestamp of the event in milliseconds since boot.

Definition at line 4 of file [event_manager.h](#).

8.45.2.3 group

```
uint8_t group
```

Event group.

Definition at line 6 of file [event_manager.h](#).

8.45.2.4 event

```
uint8_t event
```

Event code.

Definition at line 8 of file [event_manager.h](#).

8.46 event_manager.h

[Go to the documentation of this file.](#)

```

00001
00017
00018 #ifndef EVENT_MANAGER_H
00019 #define EVENT_MANAGER_H
00020
00021 #include "PowerManager.h"
00022 #include <stdint>
00023 #include <string>
00024 #include "pico/mutex.h"
00025 #include "storage.h"
00026 #include "utils.h"
00027 #include "system_state_manager.h"
00028
00032 #define EVENT_BUFFER_SIZE 100
00033
00037 #define EVENT_FLUSH_THRESHOLD 10
00038
00042 #define EVENT_LOG_FILE "/event_log.csv"
00043
00044
00050 enum class EventGroup : uint8_t {
00052     SYSTEM = 0x00,
00054     POWER = 0x01,
00056     COMMS = 0x02,
00058     GPS = 0x03,
00060     CLOCK = 0x04
00061 };
00062
00063
00069 enum class SystemEvent : uint8_t {
00071     BOOT = 0x01,
00073     SHUTDOWN = 0x02,
00075     WATCHDOG_RESET = 0x03,
00077     CORE1_START = 0x04,
00079     CORE1_STOP = 0x05
00080 };
00081
00087 enum class PowerEvent : uint8_t {
00089     BATTERY_LOW = 0x01,
00091     BATTERY_FULL = 0x02,
00093     POWER_FALLING = 0x03,
00095     BATTERY_NORMAL = 0x04,
00097     SOLAR_ACTIVE = 0x05,
00099     SOLAR_INACTIVE = 0x06,
00101     USB_CONNECTED = 0x07,
00103     USB_DISCONNECTED = 0x08
00104 };
00105
00106
00112 enum class CommsEvent : uint8_t {
00114     RADIO_INIT = 0x01,
00116     RADIO_ERROR = 0x02,
00118     MSG_RECEIVED = 0x03,
00120     MSG_SENT = 0x04,
00122     UART_ERROR = 0x06
00123 };
00124
00130 enum class GPSEvent : uint8_t {
00132     LOCK = 0x01,
00134     LOST = 0x02,
00136     ERROR = 0x03,
00138     POWER_ON = 0x04,
00140     POWER_OFF = 0x05,
00142     DATA_READY = 0x06,
00144     PASS_THROUGH_START = 0x07,
00146     PASS_THROUGH_END = 0x08
00147 };
00148
00154 enum class ClockEvent : uint8_t {
00156     CHANGED = 0x01,
00158     GPS_SYNC = 0x02,
00160     GPS_SYNC_DATA_NOT_READY = 0x03
00161 };
00162
00163
00169 class EventLog {
00170     public:
00172         uint16_t id;
00174         uint32_t timestamp;
00176         uint8_t group;
00178         uint8_t event;
00179         } __attribute__((packed));

```

```

00180
00181
00190 class EventManager {
00191 private:
00192     EventLog events[EVENT_BUFFER_SIZE];
00193     size_t eventCount;
00194     size_t writeIndex;
00195     mutex_t eventMutex;
00196     uint16_t nextEventId;
00197     size_t eventsSinceFlush;
00198
00199     EventManager() :
00200         eventCount(0),
00201         writeIndex(0),
00202         nextEventId(0),
00203         eventsSinceFlush(0)
00204     {
00205         mutex_init(&eventMutex);
00206     }
00207
00208     EventManager(const EventManager&) = delete;
00209     EventManager& operator=(const EventManager&) = delete;
00210
00211 public:
00216     static EventManager& get_instance() {
00217         static EventManager instance;
00218         return instance;
00219     }
00220
00225     bool init();
00226
00232     void log_event(uint8_t group, uint8_t event);
00233
00239     const EventLog& get_event(size_t index) const;
00240
00245     size_t get_event_count() const { return eventCount; }
00246
00251     bool save_to_storage();
00252
00257     bool load_from_storage();
00258 };
00259
00266 class EventEmitter {
00267 public:
00274     template<typename T>
00275     static void emit(EventGroup group, T event) {
00276         EventManager::get_instance().log_event(
00277             static_cast<uint8_t>(group),
00278             static_cast<uint8_t>(event)
00279         );
00280     }
00281 };
00282
00283 #endif // EVENT_MANAGER_H

```

8.47 lib/location/gps_collector.cpp File Reference

Implementation of the GPS data collector module.

```

#include "lib/location/gps_collector.h"
#include "utils.h"
#include "pico/time.h"
#include "lib/location/NMEA/nmea_data.h"
#include "event_manager.h"
#include <vector>
#include <ctime>
#include <cstring>
#include "DS3231.h"
#include <sstream>
#include "system_state_manager.h"

```

Macros

- `#define MAX_RAW_DATA_LENGTH 256`
Maximum length of the raw data buffer for NMEA sentences.

Functions

- `std::vector< std::string > splitString (const std::string &str, char delimiter)`
Splits a string into tokens based on a delimiter.
- `void collect_gps_data ()`
Collects GPS data from the UART and updates the NMEA data.

8.47.1 Detailed Description

Implementation of the GPS data collector module.

This file implements the function `collect_gps_data`, which is responsible for reading raw NMEA sentences from the GPS UART, parsing them, and updating the NMEA data in the `NMEADData` singleton.

Definition in file `gps_collector.cpp`.

8.48 gps_collector.cpp

[Go to the documentation of this file.](#)

```

00001
00014
00015 #include "lib/location/gps_collector.h"
00016 #include "utils.h"
00017 #include "pico/time.h"
00018 #include "lib/location/NMEA/nmea_data.h"
00019 #include "event_manager.h"
00020 #include <vector>
00021 #include <ctime>
00022 #include <cstring>
00023 #include "DS3231.h"
00024 #include <sstream>
00025 #include "system_state_manager.h"
00026
00030 #define MAX_RAW_DATA_LENGTH 256
00031
00040 std::vector<std::string> splitString(const std::string& str, char delimiter) {
00041     std::vector<std::string> tokens;
00042     std::stringstream ss(str);
00043     std::string token;
00044     while (std::getline(ss, token, delimiter)) {
00045         tokens.push_back(token);
00046     }
00047     return tokens;
00048 }
00049
00059 void collect_gps_data() {
00060
00061     if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
00062         return;
00063     }
00064
00065     static char raw_data_buffer[MAX_RAW_DATA_LENGTH];
00066     static int raw_data_index = 0;
00067
00068     while (uart_is_readable(GPS_UART_PORT)) {
00069         char c = uart_getc(GPS_UART_PORT);
00070
00071         if (c == '\r' || c == '\n') {
00072             // End of message
00073             if (raw_data_index > 0) {
00074                 raw_data_buffer[raw_data_index] = '\0';

```

```

00075         std::string message(raw_data_buffer);
00076         raw_data_index = 0;
00077
00078         // Split the message into tokens
00079         std::vector<std::string> tokens = splitString(message, ',');
00080
00081         // Update the global vectors based on the sentence type
00082         if (message.find("$GPRMC") == 0) {
00083             NMEAData::get_instance().update_rmc_tokens(tokens);
00084         } else if (message.find("$GPGLGA") == 0) {
00085             NMEAData::get_instance().update_gga_tokens(tokens);
00086         }
00087     }
00088 } else {
00089     // Append to buffer
00090     if (raw_data_index < MAX_RAW_DATA_LENGTH - 1) {
00091         raw_data_buffer[raw_data_index++] = c;
00092     } else {
00093         raw_data_index = 0;
00094     }
00095 }
00096 }
00097 }

```

8.49 lib/location/gps_collector.h File Reference

Header file for the GPS data collector module.

```

#include <string>
#include "hardware/uart.h"
#include "lib/location/NMEA/nmea_data.h"
#include "pin_config.h"

```

Functions

- void [collect_gps_data](#) ()
Collects GPS data from the UART and updates the NMEA data.

8.49.1 Detailed Description

Header file for the GPS data collector module.

This file defines the function `collect_gps_data`, which is responsible for reading raw NMEA sentences from the GPS UART, parsing them, and updating the NMEA data in the [NMEAData](#) singleton.

Definition in file [gps_collector.h](#).

8.50 gps_collector.h

[Go to the documentation of this file.](#)

```

00001
00014
00015 #ifndef GPS_COLLECTOR_H
00016 #define GPS_COLLECTOR_H
00017
00018 #include <string>
00019 #include "hardware/uart.h"
00020 #include "lib/location/NMEA/nmea_data.h" // Include the new header
00021 #include "pin_config.h"
00022
00032 void collect_gps_data();
00033
00034 #endif

```

8.51 lib/location/NMEA/NMEA_data.h File Reference

Header file for the [NMEADData](#) class, which manages parsed NMEA sentences.

```
#include <vector>
#include <string>
#include "pico/sync.h"
#include <ctime>
#include <cstring>
```

Classes

- class [NMEADData](#)
Manages parsed NMEA sentences.

8.51.1 Detailed Description

Header file for the [NMEADData](#) class, which manages parsed NMEA sentences.

This file defines the [NMEADData](#) class, a singleton that stores and provides access to parsed data from NMEA sentences received from a GPS module. It includes methods for updating and retrieving RMC and GGA tokens, as well as converting the data to a Unix timestamp.

Definition in file [NMEA_data.h](#).

8.52 NMEA_data.h

[Go to the documentation of this file.](#)

```
00001
00015
00016 #ifndef NMEA_DATA_H
00017 #define NMEA_DATA_H
00018
00019 #include <vector>
00020 #include <string>
00021 #include "pico/sync.h"
00022 #include <ctime>
00023 #include <cstring>
00024
00033 class NMEADData {
00034 private:
00036     std::vector<std::string> rmc_tokens_;
00038     std::vector<std::string> gga_tokens_;
00040     mutex_t rmc_mutex_;
00042     mutex_t gga_mutex_;
00043
00048     NMEADData() {
00049         mutex_init(&rmc_mutex_);
00050         mutex_init(&gga_mutex_);
00051     }
00052
00056     NMEADData(const NMEADData&) = delete;
00060     NMEADData& operator=(const NMEADData&) = delete;
00061
00062 public:
00067     static NMEADData& get_instance() {
00068         static NMEADData instance;
00069         return instance;
00070     }
00071
00076     void update_rmc_tokens(const std::vector<std::string>& tokens) {
00077         mutex_enter_blocking(&rmc_mutex_);
```

```

00078         rmc_tokens_ = tokens;
00079         mutex_exit(&rmc_mutex_);
00080     }
00081
00086     void update_gga_tokens(const std::vector<std::string>& tokens) {
00087         mutex_enter_blocking(&gga_mutex_);
00088         gga_tokens_ = tokens;
00089         mutex_exit(&gga_mutex_);
00090     }
00091
00096     std::vector<std::string> get_rmc_tokens() const {
00097         mutex_enter_blocking(const_cast<mutex_t*>(&rmc_mutex_));
00098         std::vector<std::string> copy = rmc_tokens_;
00099         mutex_exit(const_cast<mutex_t*>(&rmc_mutex_));
00100         return copy;
00101     }
00102
00107     std::vector<std::string> get_gga_tokens() const {
00108         mutex_enter_blocking(const_cast<mutex_t*>(&gga_mutex_));
00109         std::vector<std::string> copy = gga_tokens_;
00110         mutex_exit(const_cast<mutex_t*>(&gga_mutex_));
00111         return copy;
00112     }
00113
00118     bool has_valid_time() const {
00119         return rmc_tokens_.size() >= 10 && rmc_tokens_[1].length() > 5;
00120     }
00121
00126     time_t get_unix_time() const {
00127         if (!has_valid_time()) {
00128             return 0; // Invalid time
00129         }
00130
00131         // Parse date and time from RMC tokens
00132         // Format: hhmmss.sss,A,ddmm.mmmm,N,dddmm.mmmm,W,speed,course,ddmmyy
00133         std::string time_str = rmc_tokens_[1]; // hhmmss.sss
00134         std::string date_str = rmc_tokens_[9]; // ddmmyy
00135
00136         if (time_str.length() < 6 || date_str.length() < 6) {
00137             return 0;
00138         }
00139
00140         struct tm timeinfo;
00141         memset(&timeinfo, 0, sizeof(timeinfo));
00142
00143         // Parse time: hours (0-1), minutes (2-3), seconds (4-5)
00144         timeinfo.tm_hour = std::stoi(time_str.substr(0, 2));
00145         timeinfo.tm_min = std::stoi(time_str.substr(2, 2));
00146         timeinfo.tm_sec = std::stoi(time_str.substr(4, 2));
00147
00148         // Parse date: day (0-1), month (2-3), year (4-5)
00149         timeinfo.tm_mday = std::stoi(date_str.substr(0, 2));
00150         timeinfo.tm_mon = std::stoi(date_str.substr(2, 2)) - 1; // Month is 0-11
00151         timeinfo.tm_year = std::stoi(date_str.substr(4, 2)) + 100; // Year is since 1900
00152
00153         return mktime(&timeinfo);
00154     }
00155 };
00156
00157 #endif

```

8.53 lib/pin_config.h File Reference

```
#include <stdint.h>
```

Macros

- #define `DEBUG_UART_PORT` uart0
- #define `DEBUG_UART_BAUD_RATE` 115200
- #define `DEBUG_UART_TX_PIN` 0
- #define `DEBUG_UART_RX_PIN` 1
- #define `MAIN_I2C_PORT` i2c1
- #define `MAIN_I2C_SDA_PIN` 6

- `#define MAIN_I2C_SCL_PIN 7`
- `#define GPS_UART_PORT uart1`
- `#define GPS_UART_BAUD_RATE 9600`
- `#define GPS_UART_TX_PIN 8`
- `#define GPS_UART_RX_PIN 9`
- `#define GPS_POWER_ENABLE_PIN 14`
- `#define SENSORS_POWER_ENABLE_PIN 15`
- `#define SENSORS_I2C_PORT i2c0`
- `#define SENSORS_I2C_SDA_PIN 4`
- `#define SENSORS_I2C_SCL_PIN 5`
- `#define BUFFER_SIZE 85`
- `#define SD_SPI_PORT spi1`
- `#define SD_MISO_PIN 12`
- `#define SD_MOSI_PIN 11`
- `#define SD_SCK_PIN 10`
- `#define SD_CS_PIN 13`
- `#define SD_CARD_DETECT_PIN 28`
- `#define SX1278_MISO 16`
- `#define SX1278_CS 17`
- `#define SX1278_SCK 18`
- `#define SX1278_MOSI 19`
- `#define SPI_PORT spi0`
- `#define READ_BIT 0x80`
- `#define LORA_DEFAULT_SPI spi0`
- `#define LORA_DEFAULT_SPI_FREQUENCY 8E6`
- `#define LORA_DEFAULT_SS_PIN 17`
- `#define LORA_DEFAULT_RESET_PIN 22`
- `#define LORA_DEFAULT_DIO0_PIN 20`
- `#define PA_OUTPUT_RFO_PIN 11`
- `#define PA_OUTPUT_PA_BOOST_PIN 12`

Variables

- `constexpr int lora_cs_pin = 17`
- `constexpr int lora_reset_pin = 22`
- `constexpr int lora_irq_pin = 28`
- `uint8_t lora_address_local = 37`
- `uint8_t lora_address_remote = 21`

8.53.1 Macro Definition Documentation

8.53.1.1 DEBUG_UART_PORT

```
#define DEBUG_UART_PORT uart0
```

Definition at line 8 of file [pin_config.h](#).

8.53.1.2 DEBUG_UART_BAUD_RATE

```
#define DEBUG_UART_BAUD_RATE 115200
```

Definition at line 9 of file [pin_config.h](#).

8.53.1.3 DEBUG_UART_TX_PIN

```
#define DEBUG_UART_TX_PIN 0
```

Definition at line 11 of file [pin_config.h](#).

8.53.1.4 DEBUG_UART_RX_PIN

```
#define DEBUG_UART_RX_PIN 1
```

Definition at line 12 of file [pin_config.h](#).

8.53.1.5 MAIN_I2C_PORT

```
#define MAIN_I2C_PORT i2c1
```

Definition at line 14 of file [pin_config.h](#).

8.53.1.6 MAIN_I2C_SDA_PIN

```
#define MAIN_I2C_SDA_PIN 6
```

Definition at line 15 of file [pin_config.h](#).

8.53.1.7 MAIN_I2C_SCL_PIN

```
#define MAIN_I2C_SCL_PIN 7
```

Definition at line 16 of file [pin_config.h](#).

8.53.1.8 GPS_UART_PORT

```
#define GPS_UART_PORT uart1
```

Definition at line 19 of file [pin_config.h](#).

8.53.1.9 GPS_UART_BAUD_RATE

```
#define GPS_UART_BAUD_RATE 9600
```

Definition at line 20 of file [pin_config.h](#).

8.53.1.10 GPS_UART_TX_PIN

```
#define GPS_UART_TX_PIN 8
```

Definition at line 21 of file [pin_config.h](#).

8.53.1.11 GPS_UART_RX_PIN

```
#define GPS_UART_RX_PIN 9
```

Definition at line 22 of file [pin_config.h](#).

8.53.1.12 GPS_POWER_ENABLE_PIN

```
#define GPS_POWER_ENABLE_PIN 14
```

Definition at line 23 of file [pin_config.h](#).

8.53.1.13 SENSORS_POWER_ENABLE_PIN

```
#define SENSORS_POWER_ENABLE_PIN 15
```

Definition at line 25 of file [pin_config.h](#).

8.53.1.14 SENSORS_I2C_PORT

```
#define SENSORS_I2C_PORT i2c0
```

Definition at line 26 of file [pin_config.h](#).

8.53.1.15 SENSORS_I2C_SDA_PIN

```
#define SENSORS_I2C_SDA_PIN 4
```

Definition at line 27 of file [pin_config.h](#).

8.53.1.16 SENSORS_I2C_SCL_PIN

```
#define SENSORS_I2C_SCL_PIN 5
```

Definition at line 28 of file [pin_config.h](#).

8.53.1.17 BUFFER_SIZE

```
#define BUFFER_SIZE 85
```

Definition at line 30 of file [pin_config.h](#).

8.53.1.18 SD_SPI_PORT

```
#define SD_SPI_PORT spi1
```

Definition at line 33 of file [pin_config.h](#).

8.53.1.19 SD_MISO_PIN

```
#define SD_MISO_PIN 12
```

Definition at line 34 of file [pin_config.h](#).

8.53.1.20 SD_MOSI_PIN

```
#define SD_MOSI_PIN 11
```

Definition at line 35 of file [pin_config.h](#).

8.53.1.21 SD_SCK_PIN

```
#define SD_SCK_PIN 10
```

Definition at line 36 of file [pin_config.h](#).

8.53.1.22 SD_CS_PIN

```
#define SD_CS_PIN 13
```

Definition at line 37 of file [pin_config.h](#).

8.53.1.23 SD_CARD_DETECT_PIN

```
#define SD_CARD_DETECT_PIN 28
```

Definition at line 38 of file [pin_config.h](#).

8.53.1.24 SX1278_MISO

```
#define SX1278_MISO 16
```

Definition at line 40 of file [pin_config.h](#).

8.53.1.25 SX1278_CS

```
#define SX1278_CS 17
```

Definition at line 41 of file [pin_config.h](#).

8.53.1.26 SX1278_SCK

```
#define SX1278_SCK 18
```

Definition at line 42 of file [pin_config.h](#).

8.53.1.27 SX1278_MOSI

```
#define SX1278_MOSI 19
```

Definition at line 43 of file [pin_config.h](#).

8.53.1.28 SPI_PORT

```
#define SPI_PORT spi0
```

Definition at line 45 of file [pin_config.h](#).

8.53.1.29 READ_BIT

```
#define READ_BIT 0x80
```

Definition at line 46 of file [pin_config.h](#).

8.53.1.30 LORA_DEFAULT_SPI

```
#define LORA_DEFAULT_SPI spi0
```

Definition at line 48 of file [pin_config.h](#).

8.53.1.31 LORA_DEFAULT_SPI_FREQUENCY

```
#define LORA_DEFAULT_SPI_FREQUENCY 8E6
```

Definition at line 49 of file [pin_config.h](#).

8.53.1.32 LORA_DEFAULT_SS_PIN

```
#define LORA_DEFAULT_SS_PIN 17
```

Definition at line 50 of file [pin_config.h](#).

8.53.1.33 LORA_DEFAULT_RESET_PIN

```
#define LORA_DEFAULT_RESET_PIN 22
```

Definition at line 51 of file [pin_config.h](#).

8.53.1.34 LORA_DEFAULT_DIO0_PIN

```
#define LORA_DEFAULT_DIO0_PIN 20
```

Definition at line 52 of file [pin_config.h](#).

8.53.1.35 PA_OUTPUT_RFO_PIN

```
#define PA_OUTPUT_RFO_PIN 11
```

Definition at line 54 of file [pin_config.h](#).

8.53.1.36 PA_OUTPUT_PA_BOOST_PIN

```
#define PA_OUTPUT_PA_BOOST_PIN 12
```

Definition at line 55 of file [pin_config.h](#).

8.53.2 Variable Documentation

8.53.2.1 lora_cs_pin

```
int lora_cs_pin = 17 [inline], [constexpr]
```

Definition at line 57 of file [pin_config.h](#).

8.53.2.2 lora_reset_pin

```
int lora_reset_pin = 22 [inline], [constexpr]
```

Definition at line 58 of file [pin_config.h](#).

8.53.2.3 lora_irq_pin

```
int lora_irq_pin = 28 [inline], [constexpr]
```

Definition at line 59 of file [pin_config.h](#).

8.53.2.4 lora_address_local

```
uint8_t lora_address_local = 37 [inline]
```

Definition at line 61 of file [pin_config.h](#).

8.53.2.5 lora_address_remote

```
uint8_t lora_address_remote = 21 [inline]
```

Definition at line 62 of file [pin_config.h](#).

8.54 pin_config.h

[Go to the documentation of this file.](#)

```

00001
00002 #ifndef PIN_CONFIG_H
00003 #define PIN_CONFIG_H
00004
00005 #include <stdint.h>
00006
00007 //DEBUG uart
00008 #define DEBUG_UART_PORT uart0
00009 #define DEBUG_UART_BAUD_RATE 115200
00010
00011 #define DEBUG_UART_TX_PIN 0
00012 #define DEBUG_UART_RX_PIN 1
00013
00014 #define MAIN_I2C_PORT i2c1
00015 #define MAIN_I2C_SDA_PIN 6
00016 #define MAIN_I2C_SCL_PIN 7
00017
00018 // GPS configuration
00019 #define GPS_UART_PORT uart1
00020 #define GPS_UART_BAUD_RATE 9600
00021 #define GPS_UART_TX_PIN 8
00022 #define GPS_UART_RX_PIN 9
00023 #define GPS_POWER_ENABLE_PIN 14
00024
00025 #define SENSORS_POWER_ENABLE_PIN 15
00026 #define SENSORS_I2C_PORT i2c0
00027 #define SENSORS_I2C_SDA_PIN 4
00028 #define SENSORS_I2C_SCL_PIN 5
00029
00030 #define BUFFER_SIZE 85
00031
00032 // SPI configuration for SD card
00033 #define SD_SPI_PORT spi1
00034 #define SD_MISO_PIN 12
00035 #define SD_MOSI_PIN 11
00036 #define SD_SCK_PIN 10
00037 #define SD_CS_PIN 13
00038 #define SD_CARD_DETECT_PIN 28
00039
00040 #define SX1278_MISO 16
00041 #define SX1278_CS 17
00042 #define SX1278_SCK 18
00043 #define SX1278_MOSI 19
00044
00045 #define SPI_PORT spi0
00046 #define READ_BIT 0x80
00047
00048 #define LORA_DEFAULT_SPI spi0
00049 #define LORA_DEFAULT_SPI_FREQUENCY 8E6
00050 #define LORA_DEFAULT_SS_PIN 17
00051 #define LORA_DEFAULT_RESET_PIN 22
00052 #define LORA_DEFAULT_DIO0_PIN 20
00053
00054 #define PA_OUTPUT_RFO_PIN 11
00055 #define PA_OUTPUT_PA_BOOST_PIN 12
00056
00057 inline constexpr int lora_cs_pin = 17; // LoRa radio chip select
00058 inline constexpr int lora_reset_pin = 22; // LoRa radio reset
00059 inline constexpr int lora_irq_pin = 28; // LoRa hardware interrupt pin
00060
00061 inline uint8_t lora_address_local = 37; // address of this device
00062 inline uint8_t lora_address_remote = 21; // destination to send to
00063
00064 #endif

```

8.55 lib/powerman/INA3221/INA3221.cpp File Reference

Implementation of the [INA3221](#) power monitor driver.

```

#include "INA3221.h"
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/i2c.h"

```

```
#include <iostream>
#include "pin_config.h"
#include "utils.h"
#include <sstream>
```

8.55.1 Detailed Description

Implementation of the [INA3221](#) power monitor driver.

This file contains the implementation for the [INA3221](#) triple-channel power monitor, providing functionality for voltage, current, and power monitoring with alert capabilities.

Definition in file [INA3221.cpp](#).

8.56 INA3221.cpp

[Go to the documentation of this file.](#)

```
00001 #include "INA3221.h"
00002 #include <stdio.h>
00003 #include "pico/stdlib.h"
00004 #include "hardware/i2c.h"
00005 #include <iostream>
00006 #include "pin_config.h"
00007 #include "utils.h"
00008 #include <sstream>
00009
00010
00017
00018
00033
00034
00041 INA3221::INA3221(ina3221_addr_t addr, i2c_inst_t* i2c)
00042     : _i2c_addr(addr), _i2c(i2c) {}
00043
00044
00051 bool INA3221::begin() {
00052     uart_print("INA3221 initializing...", VerbosityLevel::DEBUG);
00053
00054     _shuntRes[0] = 10;
00055     _shuntRes[1] = 10;
00056     _shuntRes[2] = 10;
00057
00058     _filterRes[0] = 10;
00059     _filterRes[1] = 10;
00060     _filterRes[2] = 10;
00061
00062     uint16_t manuf_id = get_manufacturer_id();
00063     uint16_t die_id = get_die_id();
00064     std::stringstream ss;
00065     ss << "INA3221 Manufacturer ID: 0x" << std::hex << manuf_id
00066         << ", Die ID: 0x" << die_id;
00067     uart_print(ss.str(), VerbosityLevel::INFO);
00068
00069     if (manuf_id == 0x5449 && die_id == 0x3220) {
00070         uart_print("INA3221 found and initialized.", VerbosityLevel::DEBUG);
00071         return true;
00072     } else {
00073         uart_print("INA3221 initialization failed. Incorrect IDs.", VerbosityLevel::ERROR);
00074         return false;
00075     }
00076 }
00077
00078
00084 void INA3221::reset() {
00085     conf_reg_t conf_reg;
00086
00087     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00088     conf_reg.reset = 1;
00089     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00090 }
```



```
00091
00092
00098 uint16_t INA3221::get_manufacturer_id() {
00099     uint16_t id = 0;
00100     _read(INA3221_REG_MANUF_ID, &id);
00101     return id;
00102 }
00103
00104
00110 uint16_t INA3221::get_die_id() {
00111     uint16_t id = 0;
00112     _read(INA3221_REG_DIE_ID, &id);
00113     return id;
00114 }
00115
00116
00123 uint16_t INA3221::read_register(ina3221_reg_t reg){
00124     uint16_t val = 0;
00125     _read(reg, &val);
00126     return val;
00127 }
00128
00129
00130 //configure
00131
00137 void INA3221::set_mode_power_down(){
00138     conf_reg_t conf_reg;
00139
00140     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00141     conf_reg.mode_bus_en = 0;
00142     conf_reg.mode_continuous_en = 0;
00143     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00144 }
00145
00146
00152 void INA3221::set_mode_continuous(){
00153     conf_reg_t conf_reg;
00154
00155     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00156     conf_reg.mode_continuous_en = 1;
00157     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00158 }
00159
00160
00166 void INA3221::set_mode_triggered(){
00167     conf_reg_t conf_reg;
00168
00169     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00170     conf_reg.mode_continuous_en = 0;
00171     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00172 }
00173
00174
00179 void INA3221::set_shunt_measurement_enable(){
00180     conf_reg_t conf_reg;
00181
00182     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00183     conf_reg.mode_shunt_en = 1;
00184     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00185 }
00186
00187
00192 void INA3221::set_shunt_measurement_disable(){
00193     conf_reg_t conf_reg;
00194
00195     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00196     conf_reg.mode_shunt_en = 0;
00197     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00198 }
00199
00200
00205 void INA3221::set_bus_measurement_enable(){
00206     conf_reg_t conf_reg;
00207
00208     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00209     conf_reg.mode_bus_en = 1;
00210     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00211 }
00212
00213
00218 void INA3221::set_bus_measurement_disable(){
00219     conf_reg_t conf_reg;
00220
00221     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00222     conf_reg.mode_bus_en = 0;
00223     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00224 }
```

```

00225
00226
00232 void INA3221::set_averaging_mode(ina3221_avg_mode_t mode) {
00233     conf_reg_t conf_reg;
00234
00235     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00236     conf_reg.avg_mode = mode;
00237     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00238 }
00239
00240
00246 void INA3221::set_bus_conversion_time(ina3221_conv_time_t convTime) {
00247     conf_reg_t conf_reg;
00248
00249     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00250     conf_reg.bus_conv_time = convTime;
00251     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00252 }
00253
00254
00260 void INA3221::set_shunt_conversion_time(ina3221_conv_time_t convTime) {
00261     conf_reg_t conf_reg;
00262
00263     _read(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00264     conf_reg.shunt_conv_time = convTime;
00265     _write(INA3221_REG_CONF, (uint16_t*)&conf_reg);
00266 }
00267
00268
00269 //get measurement
00276 int32_t INA3221::get_shunt_voltage(ina3221_ch_t channel) {
00277     int32_t res;
00278     ina3221_reg_t reg;
00279     uint16_t val_raw = 0;
00280
00281     switch(channel){
00282         case INA3221_CH1:
00283             reg = INA3221_REG_CH1_SHUNTV;
00284             break;
00285         case INA3221_CH2:
00286             reg = INA3221_REG_CH2_SHUNTV;
00287             break;
00288         case INA3221_CH3:
00289             reg = INA3221_REG_CH3_SHUNTV;
00290             break;
00291     }
00292
00293     _read(reg, &val_raw);
00294
00295     res = (int16_t) (val_raw » 3);
00296     res *= SHUNT_VOLTAGE_LSB_UV;
00297
00298     return res;
00299 }
00300
00301
00308 float INA3221::get_current_ma(ina3221_ch_t channel) {
00309     int32_t shunt_uV = 0;
00310     float current_A = 0;
00311
00312     shunt_uV = get_shunt_voltage(channel);
00313     current_A = shunt_uV / (int32_t)_shuntRes[channel];
00314     return current_A;
00315 }
00316
00317
00324 float INA3221::get_voltage(ina3221_ch_t channel) {
00325     float voltage_V = 0.0;
00326     ina3221_reg_t reg;
00327     uint16_t val_raw = 0;
00328
00329     switch(channel){
00330         case INA3221_CH1:
00331             reg = INA3221_REG_CH1_BUSV;
00332             break;
00333         case INA3221_CH2:
00334             reg = INA3221_REG_CH2_BUSV;
00335             break;
00336         case INA3221_CH3:
00337             reg = INA3221_REG_CH3_BUSV;
00338             break;
00339     }
00340
00341     _read(reg, &val_raw);
00342     voltage_V = val_raw / 1000.0;
00343     return voltage_V;
00344 }

```

```

00345
00346
00347 // private
00354 void INA3221::_read(ina3221_reg_t reg, uint16_t *val) {
00355     uint8_t reg_buf = reg;
00356     uint8_t data[2];
00357
00358     int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, &reg_buf, 1, true);
00359     if (ret != 1) {
00360         std::cerr << "Failed to write register address to I2C device." << std::endl;
00361         return;
00362     }
00363
00364     ret = i2c_read_blocking(MAIN_I2C_PORT, _i2c_addr, data, 2, false);
00365     if (ret != 2) {
00366         std::cerr << "Failed to read data from I2C device." << std::endl;
00367         return;
00368     }
00369
00370     *val = (data[0] << 8) | data[1];
00371 }
00372
00373
00380 void INA3221::_write(ina3221_reg_t reg, uint16_t *val) {
00381     uint8_t buf[3];
00382     buf[0] = reg;
00383     buf[1] = (*val >> 8) & 0xFF; // MSB
00384     buf[2] = (*val) & 0xFF;     // LSB
00385
00386     int ret = i2c_write_blocking(MAIN_I2C_PORT, _i2c_addr, buf, 3, false);
00387     if (ret != 3) {
00388         std::cerr << "Failed to write data to I2C device." << std::endl;
00389     }
00390 }

```

8.57 lib/powerman/INA3221/INA3221.h File Reference

Header file for the [INA3221](#) triple-channel power monitor driver.

```

#include <stdint.h>
#include <iostream>
#include <hardware/i2c.h>

```

Classes

- class [INA3221](#)
INA3221 Triple-Channel Power Monitor driver class.
- struct [INA3221::conf_reg_t](#)
Configuration register bit fields.
- struct [INA3221::masken_reg_t](#)
Mask/Enable register bit fields.

Enumerations

- enum [ina3221_addr_t](#) { [INA3221_ADDR40_GND](#) = 0b1000000 , [INA3221_ADDR41_VCC](#) = 0b1000001 , [INA3221_ADDR42_SDA](#) = 0b1000010 , [INA3221_ADDR43_SCL](#) = 0b1000011 }
- enum [ina3221_ch_t](#) { [INA3221_CH1](#) = 0 , [INA3221_CH2](#) , [INA3221_CH3](#) }

- enum `ina3221_reg_t` {
`INA3221_REG_CONF = 0`, `INA3221_REG_CH1_SHUNTV`, `INA3221_REG_CH1_BUSV`, `INA3221_REG_CH2_SHUNTV`,
`INA3221_REG_CH2_BUSV`, `INA3221_REG_CH3_SHUNTV`, `INA3221_REG_CH3_BUSV`, `INA3221_REG_CH1_CRIT_ALERT`,
`INA3221_REG_CH1_WARNING_ALERT_LIM`, `INA3221_REG_CH2_CRIT_ALERT_LIM`, `INA3221_REG_CH2_WARNING_ALERT_LIM`,
`INA3221_REG_CH3_CRIT_ALERT_LIM`,
`INA3221_REG_CH3_WARNING_ALERT_LIM`, `INA3221_REG_SHUNTV_SUM`, `INA3221_REG_SHUNTV_SUM_LIM`,
`INA3221_REG_MASK_ENABLE`,
`INA3221_REG_PWR_VALID_HI_LIM`, `INA3221_REG_PWR_VALID_LO_LIM`, `INA3221_REG_MANUF_ID`
`= 0xFE`, `INA3221_REG_DIE_ID = 0xFF` }
Register addresses for INA3221.
- enum `ina3221_conv_time_t` {
`INA3221_REG_CONF_CT_140US = 0`, `INA3221_REG_CONF_CT_204US`, `INA3221_REG_CONF_CT_332US`,
`INA3221_REG_CONF_CT_588US`,
`INA3221_REG_CONF_CT_1100US`, `INA3221_REG_CONF_CT_2116US`, `INA3221_REG_CONF_CT_4156US`,
`INA3221_REG_CONF_CT_8244US` }
Conversion time settings.
- enum `ina3221_avg_mode_t` {
`INA3221_REG_CONF_AVG_1 = 0`, `INA3221_REG_CONF_AVG_4`, `INA3221_REG_CONF_AVG_16`,
`INA3221_REG_CONF_AVG_64`,
`INA3221_REG_CONF_AVG_128`, `INA3221_REG_CONF_AVG_256`, `INA3221_REG_CONF_AVG_512`,
`INA3221_REG_CONF_AVG_1024` }
Averaging mode settings.

Variables

- const int `INA3221_CH_NUM` = 3
Number of channels in INA3221.
- const int `SHUNT_VOLTAGE_LSB_UV` = 5
LSB value for shunt voltage measurements in microvolts.

8.57.1 Detailed Description

Header file for the [INA3221](#) triple-channel power monitor driver.

Definition in file [INA3221.h](#).

8.57.2 Enumeration Type Documentation

8.57.2.1 `ina3221_addr_t`

enum `ina3221_addr_t`

Enumerator

<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](#).

8.57.2.2 ina3221_ch_t

enum [ina3221_ch_t](#)

Enumerator

INA3221_CH1	
INA3221_CH2	
INA3221_CH3	

Definition at line 23 of file [INA3221.h](#).

8.57.2.3 ina3221_reg_t

```
enum ina3221_reg_t
```

Register addresses for [INA3221](#).

Enumerator

INA3221_REG_CONF	
INA3221_REG_CH1_SHUNTV	
INA3221_REG_CH1_BUSV	
INA3221_REG_CH2_SHUNTV	
INA3221_REG_CH2_BUSV	
INA3221_REG_CH3_SHUNTV	
INA3221_REG_CH3_BUSV	
INA3221_REG_CH1_CRIT_ALERT_LIM	
INA3221_REG_CH1_WARNING_ALERT_LIM	
INA3221_REG_CH2_CRIT_ALERT_LIM	
INA3221_REG_CH2_WARNING_ALERT_LIM	
INA3221_REG_CH3_CRIT_ALERT_LIM	
INA3221_REG_CH3_WARNING_ALERT_LIM	
INA3221_REG_SHUNTV_SUM	
INA3221_REG_SHUNTV_SUM_LIM	
INA3221_REG_MASK_ENABLE	
INA3221_REG_PWR_VALID_HI_LIM	
INA3221_REG_PWR_VALID_LO_LIM	
INA3221_REG_MANUF_ID	
INA3221_REG_DIE_ID	

Definition at line 38 of file [INA3221.h](#).

8.57.2.4 ina3221_conv_time_t

```
enum ina3221_conv_time_t
```

Conversion time settings.

Time taken for each measurement conversion

Enumerator

INA3221_REG_CONF_CT_140US	
INA3221_REG_CONF_CT_204US	
INA3221_REG_CONF_CT_332US	
INA3221_REG_CONF_CT_588US	
INA3221_REG_CONF_CT_1100US	
INA3221_REG_CONF_CT_2116US	
INA3221_REG_CONF_CT_4156US	
INA3221_REG_CONF_CT_8244US	

Definition at line 65 of file [INA3221.h](#).

8.57.2.5 ina3221_avg_mode_t

```
enum ina3221_avg_mode_t
```

Averaging mode settings.

Number of samples to average for each measurement

Enumerator

INA3221_REG_CONF_AVG_1	
INA3221_REG_CONF_AVG_4	
INA3221_REG_CONF_AVG_16	
INA3221_REG_CONF_AVG_64	
INA3221_REG_CONF_AVG_128	
INA3221_REG_CONF_AVG_256	
INA3221_REG_CONF_AVG_512	
INA3221_REG_CONF_AVG_1024	

Definition at line 80 of file [INA3221.h](#).

8.57.3 Variable Documentation

8.57.3.1 INA3221_CH_NUM

```
const int INA3221_CH_NUM = 3
```

Number of channels in [INA3221](#).

Definition at line 30 of file [INA3221.h](#).

8.57.3.2 SHUNT_VOLTAGE_LSB_UV

```
const int SHUNT_VOLTAGE_LSB_UV = 5
```

LSB value for shunt voltage measurements in microvolts.

Definition at line 32 of file [INA3221.h](#).

8.58 INA3221.h

[Go to the documentation of this file.](#)

```

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_continuous_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();
00201

```

```

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
00218 #endif

```

8.59 lib/powerman/PowerManager.cpp File Reference

Implementation of the [PowerManager](#) class, which manages power-related functions.

```

#include "PowerManager.h"
#include <iostream>
#include <sstream>
#include "event_manager.h"

```

8.59.1 Detailed Description

Implementation of the [PowerManager](#) class, which manages power-related functions.

This file implements the [PowerManager](#) class, a singleton that provides methods for reading voltage and current values, configuring the [INA3221](#) power monitor, and checking power alerts.

Definition in file [PowerManager.cpp](#).

8.60 PowerManager.cpp

[Go to the documentation of this file.](#)

```

00001
00014
00015 #include "PowerManager.h"
00016 #include <iostream>
00017 #include <sstream>
00018 #include "event_manager.h"
00019
00025 PowerManager::PowerManager()
00026 : ina3221_(INA3221_ADDR40_GND, MAIN_I2C_PORT) {
00027     recursive_mutex_init(&powerman_mutex_);
00028 }
00029
00035 PowerManager& PowerManager::get_instance() {
00036     static PowerManager instance;
00037     return instance;
00038 }
00039
00045 bool PowerManager::initialize() {
00046     recursive_mutex_enter_blocking(&powerman_mutex_);
00047     initialized_ = ina3221_.begin();
00048
00049     recursive_mutex_exit(&powerman_mutex_);
00050     return initialized_;
00051 }

```

```

00052
00058 std::string PowerManager::read_device_ids() {
00059     if (!initialized_) return "noinit";
00060     recursive_mutex_enter_blocking(&powerman_mutex_);
00061     std::stringstream man_ss;
00062     man_ss << std::hex << ina3221_.get_manufacturer_id();
00063     std::string MAN = "MAN 0x" + man_ss.str();
00064
00065     std::stringstream die_ss;
00066     die_ss << std::hex << ina3221_.get_die_id();
00067     std::string DIE = "DIE 0x" + die_ss.str();
00068     recursive_mutex_exit(&powerman_mutex_);
00069     return MAN + " - " + DIE;
00070 }
00071
00077 float PowerManager::get_voltage_battery() {
00078     if (!initialized_) return 0.0f;
00079     recursive_mutex_enter_blocking(&powerman_mutex_);
00080     float voltage = ina3221_.get_voltage(INA3221_CH1);
00081     recursive_mutex_exit(&powerman_mutex_);
00082     return voltage;
00083 }
00084
00090 float PowerManager::get_voltage_5v() {
00091     if (!initialized_) return 0.0f;
00092     recursive_mutex_enter_blocking(&powerman_mutex_);
00093     float voltage = ina3221_.get_voltage(INA3221_CH2);
00094     recursive_mutex_exit(&powerman_mutex_);
00095     return voltage;
00096 }
00097
00103 float PowerManager::get_current_charge_usb() {
00104     if (!initialized_) return 0.0f;
00105     recursive_mutex_enter_blocking(&powerman_mutex_);
00106     float current = ina3221_.get_current_ma(INA3221_CH1);
00107     recursive_mutex_exit(&powerman_mutex_);
00108     return current;
00109 }
00110
00116 float PowerManager::get_current_draw() {
00117     if (!initialized_) return 0.0f;
00118     recursive_mutex_enter_blocking(&powerman_mutex_);
00119     float current = ina3221_.get_current_ma(INA3221_CH2);
00120     recursive_mutex_exit(&powerman_mutex_);
00121     return current;
00122 }
00123
00129 float PowerManager::get_current_charge_solar() {
00130     if (!initialized_) return 0.0f;
00131     recursive_mutex_enter_blocking(&powerman_mutex_);
00132     float current = ina3221_.get_current_ma(INA3221_CH3);
00133     recursive_mutex_exit(&powerman_mutex_);
00134     return current;
00135 }
00136
00142 float PowerManager::get_current_charge_total() {
00143     if (!initialized_) return 0.0f;
00144     recursive_mutex_enter_blocking(&powerman_mutex_);
00145     float current = ina3221_.get_current_ma(INA3221_CH1) + ina3221_.get_current_ma(INA3221_CH3);
00146     recursive_mutex_exit(&powerman_mutex_);
00147     return current;
00148 }
00149
00155 void PowerManager::configure(const std::map<std::string, std::string>& config) {
00156     if (!initialized_) return;
00157     recursive_mutex_enter_blocking(&powerman_mutex_);
00158
00159     if (config.find("operating_mode") != config.end()) {
00160         if (config.at("operating_mode") == "continuous") {
00161             ina3221_.set_mode_continuous();
00162         }
00163     }
00164
00165     if (config.find("averaging_mode") != config.end()) {
00166         int avg_mode = std::stoi(config.at("averaging_mode"));
00167         switch(avg_mode) {
00168             case 1:
00169                 ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_1);
00170                 break;
00171             case 4:
00172                 ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_4);
00173                 break;
00174             case 16:
00175                 ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_16);
00176                 break;
00177             default:
00178                 ina3221_.set_averaging_mode(INA3221_REG_CONF_AVG_16);

```

```

00179     }
00180 }
00181 recursive_mutex_exit(&powerman_mutex_);
00182 }
00183
00189 bool PowerManager::is_charging_solar() {
00190     if (!initialized_) return false;
00191     recursive_mutex_enter_blocking(&powerman_mutex_);
00192     bool active = get_current_charge_solar() > SOLAR_CURRENT_THRESHOLD;
00193     recursive_mutex_exit(&powerman_mutex_);
00194     return active;
00195 }
00196
00202 bool PowerManager::is_charging_usb() {
00203     if (!initialized_) return false;
00204     recursive_mutex_enter_blocking(&powerman_mutex_);
00205     bool connected = get_current_charge_usb() > USB_CURRENT_THRESHOLD;
00206     recursive_mutex_exit(&powerman_mutex_);
00207     return connected;
00208 }
00209

```

8.61 lib/powerman/PowerManager.h File Reference

Header file for the [PowerManager](#) class, which manages power-related functions.

```

#include "INA3221/INA3221.h"
#include <map>
#include <string>
#include <hardware/i2c.h>
#include "pico/stdlib.h"
#include "pico/mutex.h"

```

Classes

- class [PowerManager](#)
Manages power-related functions.

8.61.1 Detailed Description

Header file for the [PowerManager](#) class, which manages power-related functions.

This file defines the [PowerManager](#) class, a singleton that provides methods for reading voltage and current values, configuring the [INA3221](#) power monitor, and checking power alerts.

Definition in file [PowerManager.h](#).

8.62 PowerManager.h

[Go to the documentation of this file.](#)

```

00001
00014
00015 #ifndef POWER_MANAGER_H
00016 #define POWER_MANAGER_H
00017
00018 #include "INA3221/INA3221.h"
00019 #include <map>
00020 #include <string>
00021 #include <hardware/i2c.h>
00022 #include "pico/stdlib.h"
00023 #include "pico/mutex.h"
00024
00032 class PowerManager {
00033 public:
00038     PowerManager(i2c_inst_t* i2c);
00039
00044     static PowerManager& get_instance();
00045
00050     bool initialize();
00051
00056     std::string read_device_ids();
00057
00062     float get_current_charge_solar();
00063
00068     float get_current_charge_usb();
00069
00074     float get_current_charge_total();
00075
00080     float get_current_draw();
00081
00086     float get_voltage_battery();
00087
00092     float get_voltage_5v();
00093
00098     void configure(const std::map<std::string, std::string>& config);
00099
00104     bool is_charging_solar();
00105
00110     bool is_charging_usb();
00111
00112
00114     static constexpr float SOLAR_CURRENT_THRESHOLD = 50.0f; // mA
00116     static constexpr float USB_CURRENT_THRESHOLD = 50.0f; // mA
00118     static constexpr float BATTERY_LOW_THRESHOLD = 2.8f; // V
00120     static constexpr float BATTERY_FULL_THRESHOLD = 4.2f; // V
00121
00122 private:
00124     INA3221 ina3221_;
00126     bool initialized_;
00128     recursive_mutex_t powerman_mutex_;
00130     bool charging_solar_active_ = false;
00132     bool charging_usb_active_ = false;
00133
00138     PowerManager();
00139
00143     PowerManager(const PowerManager&) = delete;
00147     PowerManager& operator=(const PowerManager&) = delete;
00148 };
00149
00150 #endif // POWER_MANAGER_H

```

8.63 lib/sensors/BH1750/BH1750.cpp File Reference

Implementation of the [BH1750](#) light sensor class.

```

#include "BH1750.h"
#include "pico/stdlib.h"
#include <stdio.h>
#include <iostream>

```

8.63.1 Detailed Description

Implementation of the [BH1750](#) light sensor class.

This file contains the implementation of the [BH1750](#) class, which provides an interface to the [BH1750](#) digital light sensor using the I2C communication protocol.

Definition in file [BH1750.cpp](#).

8.64 BH1750.cpp

[Go to the documentation of this file.](#)

```

00001
00008
00009 #include "BH1750.h"
00010 #include "pico/stdlib.h"
00011 #include <stdio.h>
00012 #include <iostream>
00013
00020 BH1750::BH1750(i2c_inst_t* i2c, uint8_t addr) : _i2c_addr(addr), i2c_port_(i2c) {}
00021
00028 bool BH1750::begin(Mode mode) {
00029     write8(static_cast<uint8_t>(Mode::POWER_ON));
00030     write8(static_cast<uint8_t>(Mode::RESET));
00031     bool config_status = configure(mode);
00032
00033     return config_status;
00034 }
00035
00042 bool BH1750::configure(Mode mode) {
00043     uint8_t modeVal = static_cast<uint8_t>(mode);
00044     switch (mode) {
00045         case Mode::UNCONFIGURED_POWER_DOWN:
00046         case Mode::POWER_ON:
00047         case Mode::RESET:
00048         case Mode::CONTINUOUS_HIGH_RES_MODE:
00049         case Mode::CONTINUOUS_HIGH_RES_MODE_2:
00050         case Mode::CONTINUOUS_LOW_RES_MODE:
00051         case Mode::ONE_TIME_HIGH_RES_MODE:
00052         case Mode::ONE_TIME_HIGH_RES_MODE_2:
00053         case Mode::ONE_TIME_LOW_RES_MODE:
00054         write8(modeVal);
00055         sleep_ms(10);
00056         return true;
00057     default:
00058         return false;
00059     }
00060 }
00061
00067 float BH1750::get_light_level() {
00068     uint8_t buffer[2];
00069     i2c_read_blocking(i2c_port_, _i2c_addr, buffer, 2, false);
00070     uint16_t level = (buffer[0] << 8) | buffer[1];
00071
00072     float lux = static_cast<float>(level) / 1.2f;
00073     return lux;
00074 }
00075
00081 void BH1750::write8(uint8_t data) {
00082     uint8_t buf[1] = {data};
00083     i2c_write_blocking(i2c_port_, _i2c_addr, buf, 1, false);
00084 }

```

8.65 lib/sensors/BH1750/BH1750.h File Reference

Header file for the [BH1750](#) light sensor class.

```
#include "hardware/i2c.h"
```

Classes

- class [BH1750](#)
Class to interface with the [BH1750](#) light sensor.

Macros

- `#define _BH1750_DEVICE_ID 0xE1`
Correct content of WHO_AM_I register (not actually used in this driver).
- `#define _BH1750_MTREG_MIN 31`
Minimum value for the MTREG register.
- `#define _BH1750_MTREG_MAX 254`
Maximum value for the MTREG register.
- `#define _BH1750_DEFAULT_MTREG 69`
Default value for the MTREG register.

8.65.1 Detailed Description

Header file for the [BH1750](#) light sensor class.

This class provides an interface to the [BH1750](#) digital light sensor using the I2C communication protocol.

Definition in file [BH1750.h](#).

8.66 BH1750.h

[Go to the documentation of this file.](#)

```

00001
00008
00009 #ifndef __BH1750_H__
00010 #define __BH1750_H__
00011
00012 #include "hardware/i2c.h"
00013
00019
00025
00031
00036 #define _BH1750_DEVICE_ID 0xE1
00037
00042 #define _BH1750_MTREG_MIN 31
00043
00048 #define _BH1750_MTREG_MAX 254
00049
00054 #define _BH1750_DEFAULT_MTREG 69
00055
00060 class BH1750 {
00061 public:
00066     enum class Mode : uint8_t {
00068         UNCONFIGURED_POWER_DOWN = 0x00,
00070         POWER_ON = 0x01,
00072         RESET = 0x07,
00074         CONTINUOUS_HIGH_RES_MODE = 0x10,
00076         CONTINUOUS_HIGH_RES_MODE_2 = 0x11,
00078         CONTINUOUS_LOW_RES_MODE = 0x13,
00080         ONE_TIME_HIGH_RES_MODE = 0x20,
00082         ONE_TIME_HIGH_RES_MODE_2 = 0x21,
00084         ONE_TIME_LOW_RES_MODE = 0x23
00085     };
00086
00092     BH1750(I2C_inst_t* i2c, uint8_t addr = 0x23);
00093
00099     bool begin(Mode mode = Mode::CONTINUOUS_HIGH_RES_MODE);
00100

```

```

00106     bool configure(Mode mode);
00107
00112     float get_light_level();
00113
00114 private:
00119     void write8(uint8_t data);
00120
00122     uint8_t _i2c_addr;
00124     i2c_inst_t* i2c_port_;
00125 };
00126
00127 #endif // __BH1750_H__

```

8.67 lib/sensors/BH1750/BH1750_WRAPPER.cpp File Reference

```

#include "BH1750_WRAPPER.h"
#include <string>
#include <iostream>

```

8.68 BH1750_WRAPPER.cpp

[Go to the documentation of this file.](#)

```

00001 #include "BH1750_WRAPPER.h"
00002 #include <string>
00003 #include <iostream>
00004
00005 BH1750Wrapper::BH1750Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {
00006     sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
00007 }
00008
00009 bool BH1750Wrapper::init() {
00010     initialized_ = sensor_.begin();
00011     return initialized_;
00012 }
00013
00014 float BH1750Wrapper::read_data(SensorDataTypeIdentifier type) {
00015     if (type == SensorDataTypeIdentifier::LIGHT_LEVEL) {
00016         return sensor_.get_light_level();
00017     }
00018     return 0.0f;
00019 }
00020
00021 bool BH1750Wrapper::is_initialized() const {
00022     return initialized_;
00023 }
00024
00025 SensorType BH1750Wrapper::get_type() const {
00026     return SensorType::LIGHT;
00027 }
00028
00029 bool BH1750Wrapper::configure(const std::map<std::string, std::string>& config) {
00030     for (const auto& [key, value] : config) {
00031         if (key == "measurement_mode") {
00032             if (value == "continuously_high_resolution") {
00033                 sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE);
00034             }
00035             else if (value == "continuously_high_resolution_2") {
00036                 sensor_.configure(BH1750::Mode::CONTINUOUS_HIGH_RES_MODE_2);
00037             }
00038             else if (value == "continuously_low_resolution") {
00039                 sensor_.configure(BH1750::Mode::CONTINUOUS_LOW_RES_MODE);
00040             }
00041             else if (value == "one_time_high_resolution") {
00042                 sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE);
00043             }
00044             else if (value == "one_time_high_resolution_2") {
00045                 sensor_.configure(BH1750::Mode::ONE_TIME_HIGH_RES_MODE_2);
00046             }
00047             else if (value == "one_time_low_resolution") {
00048                 sensor_.configure(BH1750::Mode::ONE_TIME_LOW_RES_MODE);
00049             }
00050             else {

```



```

00051         std::cerr << "[BH1750Wrapper] Unknown measurement_mode value: " << value << std::endl;
00052         return false;
00053     }
00054 }
00055 else {
00056     std::cerr << "[BH1750Wrapper] Unknown configuration key: " << key << std::endl;
00057     return false;
00058 }
00059 }
00060 return true;
00061 }

```

8.69 lib/sensors/BH1750/BH1750_WRAPPER.h File Reference

```

#include "ISensor.h"
#include "BH1750.h"
#include <map>
#include <string>

```

Classes

- class [BH1750Wrapper](#)

8.70 BH1750_WRAPPER.h

[Go to the documentation of this file.](#)

```

00001 #ifndef BH1750_WRAPPER_H
00002 #define BH1750_WRAPPER_H
00003
00004 #include "ISensor.h"
00005 #include "BH1750.h"
00006 #include <map>
00007 #include <string>
00008
00009 class BH1750Wrapper : public ISensor {
00010 private:
00011     BH1750 sensor_;
00012     bool initialized_ = false;
00013
00014 public:
00015     BH1750Wrapper(i2c_inst_t* i2c);
00016     BH1750Wrapper();
00017     int get_i2c_addr();
00018     bool init() override;
00019     float read_data(SensorDataTypeIdentifier type) override;
00020     bool is_initialized() const override;
00021     SensorType get_type() const override;
00022
00023     bool configure(const std::map<std::string, std::string>& config);
00024
00025     uint8_t get_address() const override {
00026         return 0x23;
00027     }
00028 };
00029
00030 #endif // BH1750_WRAPPER_H

```

8.71 lib/sensors/BME280/BME280.cpp File Reference

Implementation of the [BME280](#) environmental sensor class.

```
#include "BME280.h"
#include <iomanip>
#include <vector>
#include <algorithm>
#include "hardware/i2c.h"
#include "pico/binary_info.h"
#include "pico/stdlib.h"
#include "utils.h"
```

8.71.1 Detailed Description

Implementation of the [BME280](#) environmental sensor class.

This file contains the implementation of the [BME280](#) class, which provides an interface to the [BME280](#) temperature, pressure, and humidity sensor using the I2C communication protocol.

Definition in file [BME280.cpp](#).

8.72 BME280.cpp

[Go to the documentation of this file.](#)

```
00001
00009
00010 #include "BME280.h"
00011 #include <iomanip>
00012 #include <vector>
00013 #include <algorithm>
00014 #include "hardware/i2c.h"
00015 #include "pico/binary_info.h"
00016 #include "pico/stdlib.h"
00017 #include "utils.h"
00018
00024 BME280::BME280(i2c_inst_t* i2cPort, uint8_t address)
00025 : i2c_port(i2cPort), device_addr(address), calib_params{}, initialized_(false), t_fine(0) {
00026 }
00027
00032 bool BME280::init() {
00033     if (!i2c_port) {
00034         uart_print("BME280 I2C port not initialized.", VerbosityLevel::ERROR);
00035         return false;
00036     }
00037
00038     // Check device ID to confirm it's a BME280
00039     uint8_t chip_id;
00040     if (!read_register(0xD0, &chip_id)) {
00041         uart_print("Failed to read chip ID from BME280.", VerbosityLevel::ERROR);
00042         return false;
00043     }
00044
00045     if (chip_id != 0x60) {
00046         uart_print("Invalid BME280 chip ID.", VerbosityLevel::ERROR);
00047         return false;
00048     }
00049
00050     // Configure sensor
00051     if (!configure_sensor()) {
00052         uart_print("Failed to configure BME280 sensor.", VerbosityLevel::ERROR);
00053         return false;
00054     }
00055
00056     // Retrieve calibration parameters
```

```

00057     if (!get_calibration_parameters()) {
00058         uart_print("Failed to get calibration parameters from BME280.", VerbosityLevel::ERROR);
00059         return false;
00060     }
00061
00062     initialized_ = true;
00063     uart_print("BME280 initialized.", VerbosityLevel::INFO);
00064     return true;
00065 }
00066
00070 void BME280::reset() {
00071     write_register(REG_RESET, 0xB6);
00072     sleep_ms(10); // Wait for reset to complete
00073 }
00074
00082 bool BME280::read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity) {
00083     if (!initialized_) {
00084         uart_print("BME280 not initialized.", VerbosityLevel::ERROR);
00085         return false;
00086     }
00087
00088     // Define the starting register address
00089     uint8_t start_reg = REG_PRESSURE_MSB;
00090     // Total bytes to read: 3 (pressure) + 3 (temperature) + 2 (humidity) = 8
00091     uint8_t buf[8] = {0};
00092
00093     // Write the starting register address
00094     if (!write_register(start_reg, 1)) {
00095         uart_print("Failed to write to BME280.", VerbosityLevel::ERROR);
00096         return false;
00097     }
00098
00099     // Read data
00100     int ret = i2c_read_blocking(i2c_port, device_addr, buf, 8, false);
00101     if (ret != 8) {
00102         uart_print("Failed to read from BME280.", VerbosityLevel::ERROR);
00103         return false;
00104     }
00105
00106     // Combine bytes to form raw values
00107     *pressure = ((int32_t)buf[0] << 12) | ((int32_t)buf[1] << 4) | ((int32_t)(buf[2] >> 4));
00108     *temperature = ((int32_t)buf[3] << 12) | ((int32_t)buf[4] << 4) | ((int32_t)(buf[5] >> 4));
00109     *humidity = ((int32_t)buf[6] << 8) | (int32_t)buf[7];
00110
00111     return true;
00112 }
00113
00119 float BME280::convert_temperature(int32_t temp_raw) const {
00120     int32_t var1, var2;
00121     var1 = (((temp_raw >> 3) - ((int32_t)calib_params.dig_t1 << 1))) * ((int32_t)calib_params.dig_t2)
00122     >> 11;
00123     var2 = (((temp_raw >> 4) - ((int32_t)calib_params.dig_t1)) * ((temp_raw >> 4) -
00124     ((int32_t)calib_params.dig_t1))) >> 12) * ((int32_t)calib_params.dig_t3) >> 14;
00125     t_fine = var1 + var2;
00126     float T = (t_fine * 5 + 128) >> 8;
00127     return T / 100.0f;
00128 }
00129
00133 float BME280::convert_pressure(int32_t pressure_raw) const {
00134     int64_t var1, var2, p;
00135     var1 = ((int64_t)t_fine) - 128000;
00136     var2 = var1 * var1 * (int64_t)calib_params.dig_p6;
00137     var2 = var2 + ((var1 * (int64_t)calib_params.dig_p5) << 17);
00138     var2 = var2 + (((int64_t)calib_params.dig_p4) << 35);
00139     var1 = ((var1 * var1 * (int64_t)calib_params.dig_p3) >> 8) + ((var1 * (int64_t)calib_params.dig_p2)
00140     << 12);
00141     var1 = (((int64_t)1 << 47) + var1) * ((int64_t)calib_params.dig_p1) >> 33;
00142
00143     if (var1 == 0) {
00144         return 0.0f; // avoid exception caused by division by zero
00145     }
00146     p = 1048576 - pressure_raw;
00147     p = ((p << 31) - var2) * 3125 / var1;
00148     var1 = (((int64_t)calib_params.dig_p9) * (p >> 13) * (p >> 13)) >> 25;
00149     var2 = (((int64_t)calib_params.dig_p8) * p) >> 19;
00150
00151     p = ((p + var1 + var2) >> 8) + (((int64_t)calib_params.dig_p7) << 4);
00152     return (float)p / 25600.0f; // in hPa
00153 }
00154
00159 float BME280::convert_humidity(int32_t humidity_raw) const {
00160     int32_t v_x1_u32r;
00161     v_x1_u32r = t_fine - 76800;
00162     v_x1_u32r = (((humidity_raw << 14) - ((int32_t)calib_params.dig_h4 << 20) -
00163     ((int32_t)calib_params.dig_h5 * v_x1_u32r)) + 16384) >> 15) *
00164     (((((v_x1_u32r * (int32_t)calib_params.dig_h6) >> 10) * ((v_x1_u32r *
00165     (int32_t)calib_params.dig_h3) >> 11) + 32768)) >> 10) + 2097152) *

```

```

00164         (int32_t)calib_params.dig_h2 + 8192) >> 14));
00165     v_x1_u32r = std::max(v_x1_u32r, (int32_t)0);
00166     v_x1_u32r = std::min(v_x1_u32r, (int32_t)419430400);
00167     float h = v_x1_u32r >> 12;
00168     return h / 1024.0f;
00169 }
00170
00175 bool BME280::get_calibration_parameters() {
00176     // Read temperature and pressure calibration data (0x88 to 0xA1)
00177     uint8_t calib_data[NUM_CALIB_PARAMS];
00178     if (!read_register(REG_DIG_T1_LSB, calib_data, NUM_CALIB_PARAMS)) {
00179         uart_print("Failed to read calibration data from BME280.", VerbosityLevel::ERROR);
00180         return false;
00181     }
00182
00183     // Parse temperature calibration data
00184     calib_params.dig_t1 = (uint16_t)(calib_data[1] << 8 | calib_data[0]);
00185     calib_params.dig_t2 = (int16_t)(calib_data[3] << 8 | calib_data[2]);
00186     calib_params.dig_t3 = (int16_t)(calib_data[5] << 8 | calib_data[4]);
00187
00188     // Parse pressure calibration data
00189     calib_params.dig_p1 = (uint16_t)(calib_data[7] << 8 | calib_data[6]);
00190     calib_params.dig_p2 = (int16_t)(calib_data[9] << 8 | calib_data[8]);
00191     calib_params.dig_p3 = (int16_t)(calib_data[11] << 8 | calib_data[10]);
00192     calib_params.dig_p4 = (int16_t)(calib_data[13] << 8 | calib_data[12]);
00193     calib_params.dig_p5 = (int16_t)(calib_data[15] << 8 | calib_data[14]);
00194     calib_params.dig_p6 = (int16_t)(calib_data[17] << 8 | calib_data[16]);
00195     calib_params.dig_p7 = (int16_t)(calib_data[19] << 8 | calib_data[18]);
00196     calib_params.dig_p8 = (int16_t)(calib_data[21] << 8 | calib_data[20]);
00197     calib_params.dig_p9 = (int16_t)(calib_data[23] << 8 | calib_data[22]);
00198
00199     calib_params.dig_h1 = calib_data[25];
00200
00201     // Read humidity calibration data (0xE1 to 0xE7)
00202     uint8_t hum_calib_data[NUM_HUM_CALIB_PARAMS];
00203     if (!read_register(REG_DIG_H2, hum_calib_data, NUM_HUM_CALIB_PARAMS)) {
00204         uart_print("Failed to read humidity calibration data from BME280.", VerbosityLevel::ERROR);
00205         return false;
00206     }
00207
00208     // Parse humidity calibration data
00209     calib_params.dig_h2 = (int16_t)(hum_calib_data[1] << 8 | hum_calib_data[0]);
00210     calib_params.dig_h3 = hum_calib_data[2];
00211     calib_params.dig_h4 = (int16_t)((hum_calib_data[3] << 4) | (hum_calib_data[4] & 0x0F));
00212     calib_params.dig_h5 = (int16_t)((hum_calib_data[5] << 4) | (hum_calib_data[4] >> 4));
00213     calib_params.dig_h6 = (int8_t)hum_calib_data[6];
00214
00215     return true;
00216 }
00217
00222 bool BME280::configure_sensor() {
00223     // Set humidity oversampling (must be set before ctrl_meas)
00224     if (!write_register(REG_CTRL_HUM, HUMIDITY_OVERSAMPLING)) {
00225         uart_print("Failed to write CTRL_HUM to BME280.", VerbosityLevel::ERROR);
00226         return false;
00227     }
00228
00229     // Write config register
00230     if (!write_register(REG_CONFIG, 0x00)) {
00231         uart_print("Failed to write CONFIG to BME280.", VerbosityLevel::ERROR);
00232         return false;
00233     }
00234
00235     // Write ctrl_meas register
00236     if (!write_register(REG_CTRL_MEAS, NORMAL_MODE)) {
00237         uart_print("Failed to write CTRL_MEAS to BME280.", VerbosityLevel::ERROR);
00238         return false;
00239     }
00240
00241     return true;
00242 }
00243
00250 bool BME280::write_register(uint8_t reg, uint8_t value) {
00251     uint8_t buf[2] = {reg, value};
00252     int ret = i2c_write_blocking(i2c_port, device_addr, buf, 2, false);
00253     return (ret == 2);
00254 }
00255
00263 bool BME280::read_register(uint8_t reg, uint8_t* data, size_t len) {
00264     int ret = i2c_write_blocking(i2c_port, device_addr, &reg, 1, true);
00265     if (ret != 1) {
00266         return false;
00267     }
00268     ret = i2c_read_blocking(i2c_port, device_addr, data, len, false);
00269     return (static_cast<size_t>(ret) == len);
00270 }
00271

```

```

00278 bool BME280::read_register(uint8_t reg, uint8_t* data) {
00279     return read_register(reg, data, 1);
00280 }

```

8.73 lib/sensors/BME280/BME280.h File Reference

Header file for the [BME280](#) environmental sensor class.

```

#include <cstdint>
#include <iostream>
#include "hardware/i2c.h"

```

Classes

- struct [BME280CalibParam](#)
Structure to hold the [BME280](#) calibration parameters.
- class [BME280](#)
Class to interface with the [BME280](#) environmental sensor.

8.73.1 Detailed Description

Header file for the [BME280](#) environmental sensor class.

This class provides an interface to the [BME280](#) temperature, pressure, and humidity sensor using the I2C communication protocol. It includes functions for initialization, reading raw sensor data, converting raw data to physical units, and configuring the sensor's operating mode.

Definition in file [BME280.h](#).

8.74 BME280.h

[Go to the documentation of this file.](#)

```

00001
00009
00010 #ifndef BME280_H
00011 #define BME280_H
00012
00013 #include <cstdint>
00014 #include <iostream>
00015 #include "hardware/i2c.h"
00016
00023 struct BME280CalibParam {
00025     uint16_t dig_t1;
00027     int16_t dig_t2;
00029     int16_t dig_t3;
00030
00032     uint16_t dig_p1;
00034     int16_t dig_p2;
00036     int16_t dig_p3;
00038     int16_t dig_p4;
00040     int16_t dig_p5;
00042     int16_t dig_p6;
00044     int16_t dig_p7;
00046     int16_t dig_p8;
00048     int16_t dig_p9;
00049
00051     uint8_t dig_h1;

```

```

00053     int16_t    dig_h2;
00055     uint8_t    dig_h3;
00057     int16_t    dig_h4;
00059     int16_t    dig_h5;
00061     int8_t     dig_h6;
00062 };
00063
00071 class BME280 {
00072 public:
00076     enum {
00078         ADDR_SDO_LOW = 0x76,
00080         ADDR_SDO_HIGH = 0x77
00081     };
00082
00089     enum class Oversampling : uint8_t {
00091         OSR_X0 = 0x00,
00093         OSR_X1 = 0x01,
00095         OSR_X2 = 0x02,
00097         OSR_X4 = 0x03,
00099         OSR_X8 = 0x04,
00101         OSR_X16 = 0x05
00102     };
00103
00109     BME280(I2C_inst_t* i2cPort, uint8_t address = ADDR_SDO_LOW);
00110
00115     bool init();
00116
00120     void reset();
00121
00129     bool read_raw_all(int32_t* temperature, int32_t* pressure, int32_t* humidity);
00130
00136     float convert_temperature(int32_t temp_raw) const;
00137
00143     float convert_pressure(int32_t pressure_raw) const;
00144
00150     float convert_humidity(int32_t humidity_raw) const;
00151
00152 private:
00159     bool write_register(uint8_t reg, uint8_t value);
00160
00167     bool read_register(uint8_t reg, uint8_t* data);
00168
00176     bool read_register(uint8_t reg, uint8_t* data, size_t len);
00177
00182     bool configure_sensor();
00183
00188     bool get_calibration_parameters();
00189
00191     I2C_inst_t* i2c_port;
00193     uint8_t device_addr;
00194
00196     BME280CalibParam calib_params;
00197
00199     bool initialized_;
00200
00202     mutable int32_t t_fine;
00203
00207     enum {
00208         REG_CONFIG                = 0xF5,
00209         REG_CTRL_MEAS             = 0xF4,
00210         REG_CTRL_HUM              = 0xF2,
00211         REG_RESET                 = 0xE0,
00212
00213         REG_PRESSURE_MSB          = 0xF7,
00214         REG_TEMPERATURE_MSB       = 0xFA,
00215         REG_HUMIDITY_MSB          = 0xFD,
00216
00217         // Calibration Registers
00218         REG_DIG_T1_LSB            = 0x88,
00219         REG_DIG_T1_MSB            = 0x89,
00220         REG_DIG_T2_LSB            = 0x8A,
00221         REG_DIG_T2_MSB            = 0x8B,
00222         REG_DIG_T3_LSB            = 0x8C,
00223         REG_DIG_T3_MSB            = 0x8D,
00224
00225         REG_DIG_P1_LSB            = 0x8E,
00226         REG_DIG_P1_MSB            = 0x8F,
00227         REG_DIG_P2_LSB            = 0x90,
00228         REG_DIG_P2_MSB            = 0x91,
00229         REG_DIG_P3_LSB            = 0x92,
00230         REG_DIG_P3_MSB            = 0x93,
00231         REG_DIG_P4_LSB            = 0x94,
00232         REG_DIG_P4_MSB            = 0x95,
00233         REG_DIG_P5_LSB            = 0x96,
00234         REG_DIG_P5_MSB            = 0x97,
00235         REG_DIG_P6_LSB            = 0x98,
00236         REG_DIG_P6_MSB            = 0x99,

```

```

00237     REG_DIG_P7_LSB      = 0x9A,
00238     REG_DIG_P7_MSB      = 0x9B,
00239     REG_DIG_P8_LSB      = 0x9C,
00240     REG_DIG_P8_MSB      = 0x9D,
00241     REG_DIG_P9_LSB      = 0x9E,
00242     REG_DIG_P9_MSB      = 0x9F,
00243
00244     // Humidity Calibration Registers
00245     REG_DIG_H1           = 0xA1,
00246     REG_DIG_H2           = 0xE1,
00247     REG_DIG_H3           = 0xE3,
00248     REG_DIG_H4           = 0xE4,
00249     REG_DIG_H5           = 0xE5,
00250     REG_DIG_H6           = 0xE7
00251 };
00252
00253 enum {
00254     HUMIDITY_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
00255     TEMPERATURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
00256     PRESSURE_OVERSAMPLING = static_cast<uint8_t>(Oversampling::OSR_X16),
00257     NORMAL_MODE = 0xB7
00258 };
00259
00260 enum {
00261     NUM_CALIB_PARAMS = 26,
00262     NUM_HUM_CALIB_PARAMS = 7
00263 };
00264 };
00265 };
00266 };
00267 };
00268 };
00269 };
00270 };
00271 };
00272 #endif // BME280_H

```

8.75 lib/sensors/BME280/BME280_WRAPPER.cpp File Reference

```
#include "BME280_WRAPPER.h"
```

8.76 BME280_WRAPPER.cpp

[Go to the documentation of this file.](#)

```

00001 #include "BME280_WRAPPER.h"
00002
00003 BME280Wrapper::BME280Wrapper(i2c_inst_t* i2c) : sensor_(i2c) {}
00004
00005 bool BME280Wrapper::init() {
00006     initialized_ = sensor_.init();
00007     return initialized_;
00008 }
00009
00010 float BME280Wrapper::read_data(SensorDataTypeIdentifier type) {
00011     int32_t temp_raw, pressure_raw, humidity_raw;
00012     sensor_.read_raw_all(&temp_raw, &pressure_raw, &humidity_raw);
00013
00014     switch(type) {
00015         case SensorDataTypeIdentifier::TEMPERATURE:
00016             return sensor_.convert_temperature(temp_raw);
00017         case SensorDataTypeIdentifier::PRESSURE:
00018             return sensor_.convert_pressure(pressure_raw);
00019         case SensorDataTypeIdentifier::HUMIDITY:
00020             return sensor_.convert_humidity(humidity_raw);
00021         default:
00022             return 0.0f;
00023     }
00024 }
00025
00026 bool BME280Wrapper::is_initialized() const {
00027     return initialized_;
00028 }
00029
00030 SensorType BME280Wrapper::get_type() const {
00031     return SensorType::ENVIRONMENT;
00032 }
00033
00034 bool BME280Wrapper::configure([[maybe_unused]] const std::map<std::string, std::string>& config) {
00035     return true;
00036 }

```

8.77 lib/sensors/BME280/BME280_WRAPPER.h File Reference

```
#include "ISensor.h"
#include "BME280.h"
```

Classes

- class [BME280Wrapper](#)

8.78 BME280_WRAPPER.h

[Go to the documentation of this file.](#)

```
00001 // BME280_WRAPPER.h
00002 #ifndef BME280_WRAPPER_H
00003 #define BME280_WRAPPER_H
00004
00005 #include "ISensor.h"
00006 #include "BME280.h"
00007
00008 class BME280Wrapper : public ISensor {
00009 private:
00010     BME280 sensor_;
00011     bool initialized_ = false;
00012
00013 public:
00014     BME280Wrapper(i2c_inst_t* i2c);
00015
00016     bool init() override;
00017     float read_data(SensorDataTypeIdentifier type) override;
00018     bool is_initialized() const override;
00019     SensorType get_type() const override;
00020     bool configure(const std::map<std::string, std::string>& config) override;
00021
00022     uint8_t get_address() const override {
00023         return 0x76;
00024     }
00025
00026 };
00027
00028 #endif // BME280_WRAPPER_H
```

8.79 lib/sensors/ISensor.cpp File Reference

Implementation of the [ISensor](#) interface and [SensorWrapper](#) class.

```
#include "ISensor.h"
#include "lib/sensors/BH1750/BH1750_WRAPPER.h"
#include "lib/sensors/BME280/BME280_WRAPPER.h"
#include "lib/utils.h"
```

8.79.1 Detailed Description

Implementation of the [ISensor](#) interface and [SensorWrapper](#) class.

This file implements the [ISensor](#) interface and [SensorWrapper](#) class, which provide a common interface for interacting with different types of sensors.

Definition in file [ISensor.cpp](#).

8.80 ISensor.cpp

[Go to the documentation of this file.](#)

```

00001
00014
00015 #include "ISensor.h"
00016 #include "lib/sensors/BH1750/BH1750_WRAPPER.h"
00017 #include "lib/sensors/BME280/BME280_WRAPPER.h"
00018 #include "lib/Utils.h"
00019
00027 bool SensorWrapper::sensor_init(SensorType type, i2c_inst_t* i2c) {
00028     switch (type) {
00029         case SensorType::LIGHT:
00030             sensors[type] = new BH1750Wrapper(i2c);
00031             break;
00032         case SensorType::ENVIRONMENT:
00033             sensors[type] = new BME280Wrapper(i2c);
00034             break;
00035         default:
00036             return false;
00037     }
00038     return sensors[type]->init();
00039 }
00040
00048 bool SensorWrapper::sensor_configure(SensorType type, const std::map<std::string, std::string>&
    config) {
00049     if (sensors.find(type) == sensors.end()) {
00050         return false;
00051     }
00052     return sensors[type]->configure(config);
00053 }
00054
00062 float SensorWrapper::sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType) {
00063     if (sensors.find(sensorType) == sensors.end()) {
00064         return -1.0f;
00065     }
00066     return sensors[sensorType]->read_data(dataType);
00067 }
00068
00075 ISensor* SensorWrapper::get_sensor(SensorType type) {
00076     return sensors[type];
00077 }
00078
00085 std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::scan_connected_sensors(i2c_inst_t* i2c) {
00086     std::vector<std::pair<SensorType, uint8_t>> connectedSensors;
00087
00088     // Scan for BH1750 (Light Sensor)
00089     BH1750Wrapper lightSensor(i2c);
00090     if (lightSensor.init()) {
00091         connectedSensors.push_back(std::make_pair(SensorType::LIGHT, lightSensor.get_address()));
00092     }
00093
00094     // Scan for BME280 (Environment Sensor)
00095     BME280Wrapper environmentSensor(i2c);
00096     if (environmentSensor.init()) {
00097         connectedSensors.push_back(std::make_pair(SensorType::ENVIRONMENT,
            environmentSensor.get_address()));
00098     }
00099
00100     return connectedSensors;
00101 }
00102
00108 std::vector<std::pair<SensorType, uint8_t>> SensorWrapper::get_available_sensors() {
00109     std::vector<std::pair<SensorType, uint8_t>> availableSensors;
00110     for (const auto& sensorPair : sensors) {
00111         availableSensors.push_back(std::make_pair(sensorPair.first,
            sensorPair.second->get_address()));
00112     }
00113     return availableSensors;
00114 }

```

8.81 lib/sensors/ISensor.h File Reference

Header file for the [ISensor](#) interface and [SensorWrapper](#) class.

```

#include <map>
#include <string>

```

```
#include <vector>
#include <utility>
#include "hardware/i2c.h"
```

Classes

- class [ISensor](#)
Abstract base class for sensors.
- class [SensorWrapper](#)
Manages a collection of sensors.

Enumerations

- enum class [SensorType](#) : uint8_t { [SensorType::NONE](#) = 0x00 , [SensorType::LIGHT](#) = 0x01 , [SensorType::ENVIRONMENT](#) = 0x02 }
 - enum class [SensorDataTypeIdIdentifier](#) : uint8_t { [SensorDataTypeIdIdentifier::NONE](#) = 0x00 , [SensorDataTypeIdIdentifier::LIGHT_LEVEL](#) = 0x01 , [SensorDataTypeIdIdentifier::TEMPERATURE](#) = 0x02 , [SensorDataTypeIdIdentifier::HUMIDITY](#) = 0x03 , [SensorDataTypeIdIdentifier::PRESSURE](#) = 0x04 }
- Enumeration of sensor types.*
- Enumeration of sensor data type identifiers.*

8.81.1 Detailed Description

Header file for the [ISensor](#) interface and [SensorWrapper](#) class.

This file defines the [ISensor](#) interface, which provides a common interface for interacting with different types of sensors. It also defines the [SensorWrapper](#) class, which manages a collection of sensors and provides methods for initializing, configuring, and reading data from them.

Definition in file [ISensor.h](#).

8.82 ISensor.h

[Go to the documentation of this file.](#)

```
00001
00016
00017 #ifndef ISENSOR_H
00018 #define ISENSOR_H
00019
00020 #include <map>
00021 #include <string>
00022 #include <vector>
00023 #include <utility>
00024 #include "hardware/i2c.h"
00025
00031 enum class SensorType : uint8_t {
00033     NONE = 0x00,
00035     LIGHT = 0x01,
00037     ENVIRONMENT = 0x02,
00038 };
00039
00045 enum class SensorDataTypeIdIdentifier : uint8_t {
00047     NONE = 0x00,
00049     LIGHT_LEVEL = 0x01,
00051     TEMPERATURE = 0x02,
```

```

00053     HUMIDITY = 0x03,
00055     PRESSURE = 0x04,
00056 };
00057
00063 class ISensor {
00064 public:
00069     virtual ~ISensor() = default;
00070
00075     virtual bool init() = 0;
00076
00082     virtual float read_data(SensorDataTypeIdentifier type) = 0;
00083
00088     virtual bool is_initialized() const = 0;
00089
00094     virtual SensorType get_type() const = 0;
00095
00101     virtual bool configure(const std::map<std::string, std::string>& config) = 0;
00102
00107     virtual uint8_t get_address() const = 0;
00108 };
00109
00116 class SensorWrapper {
00117 public:
00122     static SensorWrapper& get_instance() {
00123         static SensorWrapper instance;
00124         return instance;
00125     }
00126
00133     bool sensor_init(SensorType type, i2c_inst_t* i2c = nullptr);
00134
00141     bool sensor_configure(SensorType type, const std::map<std::string, std::string>& config);
00142
00149     float sensor_read_data(SensorType sensorType, SensorDataTypeIdentifier dataType);
00150
00156     ISensor* get_sensor(SensorType type);
00157
00163     std::vector<std::pair<SensorType, uint8_t> scan_connected_sensors(i2c_inst_t* i2c);
00164
00169     std::vector<std::pair<SensorType, uint8_t> get_available_sensors();
00170
00171 private:
00173     std::map<SensorType, ISensor*> sensors;
00174
00178     SensorWrapper() = default;
00179 };
00180
00181 #endif

```

8.83 lib/storage/storage.cpp File Reference

Implements file system operations for the Kubisat firmware.

```

#include "storage.h"
#include "errno.h"
#include "utils.h"
#include "system_state_manager.h"

```

Functions

- bool [fs_init](#) (void)
Initializes the file system on the SD card.
- bool [fs_stop](#) (void)
Unmounts the file system from the SD card.

8.83.1 Detailed Description

Implements file system operations for the Kubisat firmware.

This file contains functions for initializing the file system, opening, writing, reading, and closing files.

Definition in file [storage.cpp](#).

8.84 storage.cpp

[Go to the documentation of this file.](#)

```

00001
00012
00013 #include "storage.h"
00014 #include "errno.h"
00015 #include "utils.h"
00016 #include "system_state_manager.h"
00017
00025 bool fs_init(void) {
00026     SystemStateManager::get_instance().set_sd_card_mounted(false);
00027     uart_print("fs_init littlefs on SD card", VerbosityLevel::DEBUG);
00028     blockdevice_t *sd = blockdevice_sd_create(SD_SPI_PORT,
00029                                               SD_MOSI_PIN,
00030                                               SD_MISO_PIN,
00031                                               SD_SCK_PIN,
00032                                               SD_CS_PIN,
00033                                               24 * MHZ,
00034                                               false);
00035     filesystem_t *fat = filesystem_fat_create();
00036
00037     std::string status_string;
00038     int err = fs_mount("/", fat, sd);
00039     if (err == -1) {
00040         status_string = "Formatting / with FAT";
00041         uart_print(status_string, VerbosityLevel::WARNING);
00042         err = fs_format(fat, sd);
00043         if (err == -1) {
00044             status_string = "fs_format error: " + std::string(strerror(errno));
00045             uart_print(status_string, VerbosityLevel::ERROR);
00046             return false;
00047         }
00048         err = fs_mount("/", fat, sd);
00049         if (err == -1) {
00050             status_string = "fs_mount error: " + std::string(strerror(errno));
00051             uart_print(status_string, VerbosityLevel::ERROR);
00052             return false;
00053         }
00054     }
00055
00056     SystemStateManager::get_instance().set_sd_card_mounted(true);
00057     return true;
00058 }
00059
00065 bool fs_stop(void) {
00066     int err = fs_unmount("/");
00067     if (err == -1) {
00068         uart_print("fs_unmount error", VerbosityLevel::ERROR);
00069         return false;
00070     }
00071     SystemStateManager::get_instance().set_sd_card_mounted(false);
00072
00073     return true;
00074 }

```

8.85 lib/storage/storage.h File Reference

Header file for file system operations on the Kubisat firmware.

```

#include <stdio.h>
#include <string.h>
#include <hardware/clocks.h>
#include <hardware/flash.h>
#include "blockdevice/flash.h"
#include "blockdevice/sd.h"
#include "filesystem/littlefs.h"
#include "filesystem/vfs.h"
#include "pin_config.h"
#include "lfs.h"
#include "filesystem/fat.h"

```

Functions

- bool [fs_init](#) (void)
Initializes the file system on the SD card.
- bool [fs_stop](#) (void)
Unmounts the file system from the SD card.

8.85.1 Detailed Description

Header file for file system operations on the Kubisat firmware.

This file defines functions for initializing, mounting, and unmounting the file system on the SD card.

Definition in file [storage.h](#).

8.86 storage.h

[Go to the documentation of this file.](#)

```
00001
00013
00014 #ifndef STORAGE_H
00015 #define STORAGE_H
00016
00017 #include <stdio.h>
00018 #include <string.h>
00019 #include <hardware/clocks.h>
00020 #include <hardware/flash.h>
00021 #include "blockdevice/flash.h"
00022 #include "blockdevice/sd.h"
00023 #include "filesystem/littlefs.h"
00024 #include "filesystem/vfs.h"
00025 #include "pin_config.h"
00026 #include "lfs.h"
00027 #include "filesystem/fat.h"
00028
00036 bool fs_init(void);
00037
00043 bool fs_stop(void);
00044
00045 #endif
```

8.87 lib/system_state_manager.h File Reference

Manages the system state of the Kubisat firmware.

```
#include <mutex>
#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
```

Classes

- class [SystemStateManager](#)
Manages the system state of the Kubisat firmware.

8.87.1 Detailed Description

Manages the system state of the Kubisat firmware.

This class is a singleton that provides methods for getting and setting various system states, such as whether a bootloader reset is pending, whether GPS collection is paused, whether the SD card is mounted, and the UART verbosity level.

Definition in file [system_state_manager.h](#).

8.88 system_state_manager.h

[Go to the documentation of this file.](#)

```

00001
00015
00016 #ifndef SYSTEM_STATE_MANAGER_H
00017 #define SYSTEM_STATE_MANAGER_H
00018
00019 #include <mutex>
00020 #include "utils.h"
00021 #include "pico/multicore.h"
00022 #include "pico/sync.h"
00023
00032 class SystemStateManager {
00033     private:
00035         bool pending_bootloader_reset;
00037         bool gps_collection_paused;
00039         bool sd_card_mounted;
00041         VerbosityLevel uart_verbosity;
00043         bool sd_card_init_status;
00045         bool radio_init_status;
00047         bool light_sensor_init_status;
00049         bool env_sensor_init_status;
00051         recursive_mutex_t mutex_;
00052
00057     SystemStateManager() :
00058         pending_bootloader_reset(false),
00059         gps_collection_paused(false),
00060         sd_card_mounted(false),
00061         uart_verbosity(VerbosityLevel::DEBUG),
00062         sd_card_init_status(false),
00063         radio_init_status(false),
00064         light_sensor_init_status(false),
00065         env_sensor_init_status(false)
00066     {
00067         recursive_mutex_init(&mutex_);
00068     }
00069
00073     SystemStateManager(const SystemStateManager&) = delete;
00077     SystemStateManager& operator=(const SystemStateManager&) = delete;
00078
00079     public:
00084     static SystemStateManager& get_instance() {
00085         static SystemStateManager instance;
00086         return instance;
00087     }
00088
00093     bool is_bootloader_reset_pending() const {
00094         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00095         bool result = pending_bootloader_reset;
00096         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00097         return result;
00098     }
00099
00104     void set_bootloader_reset_pending(bool pending) {
00105         recursive_mutex_enter_blocking(&mutex_);
00106         pending_bootloader_reset = pending;
00107         recursive_mutex_exit(&mutex_);
00108     }
00109
00114     bool is_gps_collection_paused() const {
00115         recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00116         bool result = gps_collection_paused;
00117         recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00118         return result;
00119     }

```

```

00120
00125 void set_gps_collection_paused(bool paused) {
00126     recursive_mutex_enter_blocking(&mutex_);
00127     gps_collection_paused = paused;
00128     recursive_mutex_exit(&mutex_);
00129 }
00130
00135 bool is_sd_card_mounted() const {
00136     recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00137     bool result = sd_card_mounted;
00138     recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00139     return result;
00140 }
00141
00146 void set_sd_card_mounted(bool mounted) {
00147     recursive_mutex_enter_blocking(&mutex_);
00148     sd_card_mounted = mounted;
00149     recursive_mutex_exit(&mutex_);
00150 }
00151
00156 VerboseLevel get_uart_verbosity() const {
00157     recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00158     VerboseLevel result = uart_verbosity;
00159     recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00160     return result;
00161 }
00162
00167 void set_uart_verbosity(VerboseLevel level) {
00168     recursive_mutex_enter_blocking(&mutex_);
00169     uart_verbosity = level;
00170     recursive_mutex_exit(&mutex_);
00171 }
00172
00177 bool is_radio_init_ok() const {
00178     recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00179     bool result = radio_init_status;
00180     recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00181     return result;
00182 }
00183
00188 void set_radio_init_ok(bool status) {
00189     recursive_mutex_enter_blocking(&mutex_);
00190     radio_init_status = status;
00191     recursive_mutex_exit(&mutex_);
00192 }
00193
00198 bool is_light_sensor_init_ok() const {
00199     recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00200     bool result = light_sensor_init_status;
00201     recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00202     return result;
00203 }
00204
00209 void set_light_sensor_init_ok(bool status) {
00210     recursive_mutex_enter_blocking(&mutex_);
00211     light_sensor_init_status = status;
00212     recursive_mutex_exit(&mutex_);
00213 }
00214
00219 bool is_env_sensor_init_ok() const {
00220     recursive_mutex_enter_blocking(const_cast<recursive_mutex_t*>(&mutex_));
00221     bool result = env_sensor_init_status;
00222     recursive_mutex_exit(const_cast<recursive_mutex_t*>(&mutex_));
00223     return result;
00224 }
00225
00230 void set_env_sensor_init_ok(bool status) {
00231     recursive_mutex_enter_blocking(&mutex_);
00232     env_sensor_init_status = status;
00233     recursive_mutex_exit(&mutex_);
00234 }
00235 };
00236
00237 #endif

```

8.89 lib/telemetry/telemetry_manager.cpp File Reference

Implementation of telemetry collection and storage functionality.

```

#include "telemetry_manager.h"
#include "utils.h"

```

```
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include "system_state_manager.h"
```

Macros

- `#define TELEMETRY_CSV_PATH "/telemetry.csv"`
Path to the telemetry CSV file on storage media.
- `#define SENSOR_DATA_CSV_PATH "/sensors.csv"`
Path to the sensor data CSV file on storage media.
- `#define DEFAULT_SAMPLE_INTERVAL_MS 1000`
Default interval between telemetry samples in milliseconds (2 seconds)
- `#define DEFAULT_FLUSH_THRESHOLD 10`
Default number of records to collect before flushing to storage.

8.89.1 Detailed Description

Implementation of telemetry collection and storage functionality.

Handles collecting, buffering, and persisting telemetry data from various satellite subsystems including power, sensors, and GPS

Definition in file [telemetry_manager.cpp](#).

8.90 telemetry_manager.cpp

[Go to the documentation of this file.](#)

```
00001
00009
00010 #include "telemetry_manager.h"
00011 #include "utils.h"
00012 #include "storage.h"
00013 #include "PowerManager.h"
00014 #include "ISensor.h"
00015 #include "DS3231.h"
00016 #include <deque>
00017 #include <mutex>
00018 #include <iomanip>
00019 #include <sstream>
00020 #include <cstdio>
00021 #include "communication.h"
00022 #include "system_state_manager.h"
00023
00027 #define TELEMETRY_CSV_PATH "/telemetry.csv"
00028
00032 #define SENSOR_DATA_CSV_PATH "/sensors.csv"
00033
00037 #define DEFAULT_SAMPLE_INTERVAL_MS 1000
00038
00042 #define DEFAULT_FLUSH_THRESHOLD 10
```



```

00043
00044 TelemetryManager::TelemetryManager() {}
00045
00054 bool TelemetryManager::init() {
00055     mutex_init(&telemetry_mutex);
00056     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00057         uart_print("Telemetry system initialized (storage not available)", VerbosityLevel::WARNING);
00058         return false;
00059     }
00060
00061     bool success = true;
00062
00063     FILE* telemetry_file = fopen(TELEMETRY_CSV_PATH, "w");
00064     if (!telemetry_file) {
00065         telemetry_file = fopen(TELEMETRY_CSV_PATH, "w");
00066         if (telemetry_file) {
00067             fprintf(telemetry_file, "timestamp,build,battery_v,system_v,usb_ma,solar_ma,discharge_ma,"
00068                 "gps_time,latitude,lat_dir,longitude,lon_dir,speed_knots,course_deg,date,"
00069                 "fix_quality,satellites,altitude_m\n");
00070             fclose(telemetry_file);
00071             uart_print("Created new telemetry log", VerbosityLevel::INFO);
00072         }
00073         else {
00074             uart_print("Failed to create telemetry log", VerbosityLevel::ERROR);
00075             success = false;
00076         }
00077     }
00078     else {
00079         fclose(telemetry_file);
00080     }
00081
00082     FILE* sensor_file = fopen(SENSOR_DATA_CSV_PATH, "w");
00083     if (!sensor_file) {
00084         sensor_file = fopen(SENSOR_DATA_CSV_PATH, "w");
00085         if (sensor_file) {
00086             fprintf(sensor_file, "timestamp,temperature,pressure,humidity,light\n");
00087             fclose(sensor_file);
00088             uart_print("Created new sensor data log", VerbosityLevel::INFO);
00089         }
00090         else {
00091             uart_print("Failed to create sensor data log", VerbosityLevel::ERROR);
00092             success = false;
00093         }
00094     }
00095     else {
00096         fclose(sensor_file);
00097     }
00098
00099     return success;
00100 }
00101
00102
00108 void TelemetryManager::collect_power_telemetry(TelemetryRecord& record) {
00109     record.battery_voltage = PowerManager::get_instance().get_voltage_battery();
00110     record.system_voltage = PowerManager::get_instance().get_voltage_5v();
00111     record.charge_current_usb = PowerManager::get_instance().get_current_charge_usb();
00112     record.charge_current_solar = PowerManager::get_instance().get_current_charge_solar();
00113     record.discharge_current = PowerManager::get_instance().get_current_draw();
00114 }
00115
00123 void TelemetryManager::emit_power_events(float battery_voltage, float charge_current_usb, float
charge_current_solar) {
00124     static bool usb_charging_active = false;
00125     static bool solar_charging_active = false;
00126     static bool battery_low = false;
00127     static bool battery_full = false;
00128
00129     if (charge_current_usb > PowerManager::USB_CURRENT_THRESHOLD && !usb_charging_active) {
00130         EventEmitter::emit(EventGroup::POWER, PowerEvent::USB_CONNECTED);
00131         usb_charging_active = true;
00132     }
00133     else if (charge_current_usb < PowerManager::USB_CURRENT_THRESHOLD && usb_charging_active) {
00134         EventEmitter::emit(EventGroup::POWER, PowerEvent::USB_DISCONNECTED);
00135         usb_charging_active = false;
00136     }
00137
00138     if (charge_current_solar > PowerManager::SOLAR_CURRENT_THRESHOLD && !solar_charging_active) {
00139         EventEmitter::emit(EventGroup::POWER, PowerEvent::SOLAR_ACTIVE);
00140         solar_charging_active = true;
00141     }
00142     else if (charge_current_solar < PowerManager::SOLAR_CURRENT_THRESHOLD && solar_charging_active) {
00143         EventEmitter::emit(EventGroup::POWER, PowerEvent::SOLAR_INACTIVE);
00144         solar_charging_active = false;
00145     }
00146
00147     if (battery_voltage < PowerManager::BATTERY_LOW_THRESHOLD && !battery_low) {
00148         EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_LOW);

```

```

00149         battery_low = true;
00150         battery_full = false; // Cancel overcharge event
00151     }
00152     else if (battery_voltage > PowerManager::BATTERY_FULL_THRESHOLD && !battery_full) {
00153         EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_FULL);
00154         battery_full = true;
00155         battery_low = false; // Cancel low battery event
00156     }
00157     else if (battery_voltage > PowerManager::BATTERY_LOW_THRESHOLD && battery_low) {
00158         EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_NORMAL);
00159         battery_low = false;
00160     }
00161     else if (battery_voltage < PowerManager::BATTERY_FULL_THRESHOLD && battery_full) {
00162         EventEmitter::emit(EventGroup::POWER, PowerEvent::BATTERY_NORMAL);
00163         battery_full = false;
00164     }
00165 }
00166
00172 void TelemetryManager::collect_gps_telemetry(TelemetryRecord& record) {
00173     auto& nmea_data = NMEADData::get_instance();
00174     // Get GPS RMC data
00175     std::vector<std::string> rmc_tokens = nmea_data.get_rmc_tokens();
00176     if (rmc_tokens.size() >= 12) { // RMC has at least 12 fields when complete
00177         record.time = rmc_tokens[1].substr(0, 6); // Only keep HHMMSS
00178         record.latitude = rmc_tokens[3];
00179         record.lat_dir = rmc_tokens[4];
00180         record.longitude = rmc_tokens[5];
00181         record.lon_dir = rmc_tokens[6];
00182         record.speed = rmc_tokens[7];
00183         record.course = rmc_tokens[8];
00184         record.date = rmc_tokens[9];
00185     }
00186     else {
00187         // Fill with defaults if no GPS data
00188         record.time = "";
00189         record.latitude = "";
00190         record.lat_dir = "";
00191         record.longitude = "";
00192         record.lon_dir = "";
00193         record.speed = "";
00194         record.course = "";
00195         record.date = "";
00196     }
00197
00198     // Get GPS GGA data
00199     std::vector<std::string> gga_tokens = nmea_data.get_gga_tokens();
00200     if (gga_tokens.size() >= 15) { // GGA has 15 fields when complete
00201         record.fix_quality = gga_tokens[6];
00202         record.satellites = gga_tokens[7];
00203         record.altitude = gga_tokens[9];
00204     }
00205     else {
00206         // Fill with defaults if no GPS data
00207         record.fix_quality = "";
00208         record.satellites = "";
00209         record.altitude = "";
00210     }
00211 }
00212
00218 void TelemetryManager::collect_sensor_telemetry(SensorDataRecord& sensor_record) {
00219     SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00220     sensor_record.temperature = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00221         SensorDataTypeIdentifier::TEMPERATURE);
00221     sensor_record.pressure = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00222         SensorDataTypeIdentifier::PRESSURE);
00222     sensor_record.humidity = sensor_wrapper.sensor_read_data(SensorType::ENVIRONMENT,
00223         SensorDataTypeIdentifier::HUMIDITY);
00223     sensor_record.light = sensor_wrapper.sensor_read_data(SensorType::LIGHT,
00224         SensorDataTypeIdentifier::LIGHT_LEVEL);
00224 }
00225
00233 bool TelemetryManager::collect_telemetry() {
00234     uint32_t timestamp = DS3231::get_instance().get_local_time();
00235     TelemetryRecord record;
00236     record.timestamp = timestamp;
00237     record.build_version = std::to_string(BUILD_NUMBER);
00238
00239     // Collect power telemetry and emit events
00240     collect_power_telemetry(record);
00241     emit_power_events(record.battery_voltage, record.charge_current_usb, record.charge_current_solar);
00242
00243     // Collect GPS telemetry
00244     collect_gps_telemetry(record);
00245
00246     // Collect sensor telemetry
00247     SensorDataRecord sensor_record;
00248     sensor_record.timestamp = timestamp;

```

```

00249     collect_sensor_telemetry(sensor_record);
00250
00251     mutex_enter_blocking(&telemetry_mutex);
00252
00253     telemetry_buffer[telemetry_buffer_write_index] = record;
00254     sensor_data_buffer[telemetry_buffer_write_index] = sensor_record;
00255     telemetry_buffer_write_index = (telemetry_buffer_write_index + 1) % TELEMETRY_BUFFER_SIZE;
00256     if (telemetry_buffer_count < TELEMETRY_BUFFER_SIZE) {
00257         telemetry_buffer_count++;
00258     }
00259
00260     mutex_exit(&telemetry_mutex);
00261
00262     uart_print("Telemetry collected", VerbosityLevel::DEBUG);
00263
00264     return true;
00265 }
00266
00267
00275 bool TelemetryManager::flush_telemetry() {
00276     if (!SystemStateManager::get_instance().is_sd_card_mounted()) {
00277         return false;
00278     }
00279
00280     mutex_enter_blocking(&telemetry_mutex);
00281
00282     if (telemetry_buffer_count == 0) {
00283         mutex_exit(&telemetry_mutex);
00284         return true; // Nothing to save
00285     }
00286
00287     FILE* telemetry_file = fopen(TELEMETRY_CSV_PATH, "a");
00288     FILE* sensor_file = fopen(SENSOR_DATA_CSV_PATH, "a");
00289
00290     if (!telemetry_file || !sensor_file) {
00291         uart_print("Failed to open telemetry or sensor log for writing", VerbosityLevel::ERROR);
00292         if (telemetry_file) fclose(telemetry_file);
00293         if (sensor_file) fclose(sensor_file);
00294         mutex_exit(&telemetry_mutex);
00295         return false;
00296     }
00297
00298     // Calculate start index (for circular buffer)
00299     size_t read_index = 0;
00300     if (telemetry_buffer_count == TELEMETRY_BUFFER_SIZE) {
00301         // Buffer is full, start from oldest entry
00302         read_index = telemetry_buffer_write_index;
00303     }
00304
00305     // Write all records to CSV
00306     for (size_t i = 0; i < telemetry_buffer_count; i++) {
00307         fprintf(telemetry_file, "%s\n", telemetry_buffer[read_index].to_csv().c_str());
00308         fprintf(sensor_file, "%s\n", sensor_data_buffer[read_index].to_csv().c_str());
00309         read_index = (read_index + 1) % TELEMETRY_BUFFER_SIZE;
00310     }
00311
00312     // Clear buffer after successful write
00313     telemetry_buffer_count = 0;
00314     telemetry_buffer_write_index = 0;
00315
00316     fclose(telemetry_file);
00317     fclose(sensor_file);
00318
00319     mutex_exit(&telemetry_mutex);
00320     return true;
00321 }
00322
00331 bool TelemetryManager::is_telemetry_collection_time(uint32_t current_time, uint32_t&
    last_collection_time) {
00332     if (current_time - last_collection_time >= sample_interval_ms) {
00333         last_collection_time = current_time;
00334         return true;
00335     }
00336     return false;
00337 }
00338
00339
00347 bool TelemetryManager::is_telemetry_flush_time(uint32_t& collection_counter) {
00348     if (collection_counter >= flush_threshold) {
00349         collection_counter = 0;
00350         return true;
00351     }
00352     return false;
00353 }
00354
00360 std::string TelemetryManager::get_last_telemetry_record_csv() {
00361     mutex_enter_blocking(&telemetry_mutex);

```

```

00362
00363     if (telemetry_buffer_count == 0) {
00364         mutex_exit(&telemetry_mutex);
00365         return "";
00366     }
00367
00368     TelemetryRecord last_record = get_last_telemetry_record();
00369
00370     mutex_exit(&telemetry_mutex);
00371
00372     return last_record.to_csv();
00373 }
00374
00380 std::string TelemetryManager::get_last_sensor_record_csv() {
00381     mutex_enter_blocking(&telemetry_mutex);
00382
00383     if (telemetry_buffer_count == 0) {
00384         mutex_exit(&telemetry_mutex);
00385         return "";
00386     }
00387
00388     SensorDataRecord last_record = get_last_sensor_record();
00389
00390     mutex_exit(&telemetry_mutex);
00391
00392     return last_record.to_csv();
00393 }

```

8.91 lib/telemetry/telemetry_manager.h File Reference

System telemetry collection and logging.

```

#include <cstdint>
#include <string>
#include "pico/stdlib.h"
#include "lib/location/NMEA/nmea_data.h"
#include "utils.h"
#include "storage.h"
#include "PowerManager.h"
#include "ISensor.h"
#include "DS3231.h"
#include <deque>
#include <mutex>
#include <iomanip>
#include <sstream>
#include <cstdio>
#include "communication.h"
#include <functional>

```

Classes

- struct [TelemetryRecord](#)
Structure representing a single telemetry data point.
- struct [SensorDataRecord](#)
Structure representing a single sensor data point.
- class [TelemetryManager](#)
Manages the collection, storage, and retrieval of telemetry data.

8.91.1 Detailed Description

System telemetry collection and logging.

This module handles periodic collection and storage of telemetry data from various satellite subsystems including power management, sensors (temperature, pressure, humidity, light), and GPS data.

Telemetry is collected at configurable intervals and stored in a circular buffer before being flushed to persistent storage after a configurable number of records are collected.

Definition in file [telemetry_manager.h](#).

8.92 telemetry_manager.h

[Go to the documentation of this file.](#)

```

00001
00015
00016
00017 #ifndef TELEMETRY_MANAGER_H
00018 #define TELEMETRY_MANAGER_H
00019
00020 #include <cstdint>
00021 #include <string>
00022 #include "pico/stdlib.h"
00023 #include "lib/location/NMEA/nmea_data.h"
00024 #include "utils.h"
00025 #include "storage.h"
00026 #include "PowerManager.h"
00027 #include "ISensor.h"
00028 #include "DS3231.h"
00029 #include <deque>
00030 #include <mutex>
00031 #include <iomanip>
00032 #include <sstream>
00033 #include <cstdio>
00034 #include "communication.h"
00035 #include <functional>
00036
00043 struct TelemetryRecord {
00044     uint32_t timestamp;
00045
00046     std::string build_version;
00047
00048     // Power data
00049     float battery_voltage;
00050     float system_voltage;
00051     float charge_current_usb;
00052     float charge_current_solar;
00053     float discharge_current;
00054
00055     // GPS data - key RMC fields
00056     std::string time;
00057     std::string latitude;
00058     std::string lat_dir;
00059     std::string longitude;
00060     std::string lon_dir;
00061     std::string speed;
00062     std::string course;
00063     std::string date;
00064
00065     // GPS data - key GGA fields
00066     std::string fix_quality;
00067     std::string satellites;
00068     std::string altitude;
00069
00070
00076     std::string to_csv() const {
00077         std::stringstream ss;
00078         ss << timestamp << ", "
00079             << build_version << ", "
00080             << std::fixed << std::setprecision(3)
00081             << battery_voltage << ", "
00082             << system_voltage << ", "
00083             << charge_current_usb << ", "
00084             << charge_current_solar << ", "

```

```

00085         « discharge_current « ", "
00086
00087         // GPS RMC data
00088         « time « ", "
00089         « latitude « ", " « lat_dir « ", "
00090         « longitude « ", " « lon_dir « ", "
00091         « speed « ", "
00092         « course « ", "
00093         « date « ", "
00094         // GPS GGA data
00095         « fix_quality « ", "
00096         « satellites « ", "
00097         « altitude;
00098         return ss.str();
00099     }
00100 };
00101
00102
00110 struct SensorDataRecord {
00111     uint32_t timestamp;
00112     float temperature;
00113     float pressure;
00114     float humidity;
00115     float light;
00116
00122     std::string to_csv() const {
00123         std::stringstream ss;
00124         ss « timestamp « ", "
00125           « std::fixed « std::setprecision(3)
00126           « temperature « ", "
00127           « pressure « ", "
00128           « humidity « ", "
00129           « light;
00130         return ss.str();
00131     }
00132 };
00133
00134
00145 class TelemetryManager {
00146 public:
00151     static TelemetryManager& get_instance() {
00152         static TelemetryManager instance;
00153         return instance;
00154     }
00155
00162     bool init();
00163
00170     bool collect_telemetry();
00171
00177     void collect_power_telemetry(TelemetryRecord& record);
00178
00186     void emit_power_events(float battery_voltage, float charge_current_usb, float
charge_current_solar);
00187
00193     void collect_gps_telemetry(TelemetryRecord& record);
00194
00200     void collect_sensor_telemetry(SensorDataRecord& sensor_record);
00201
00208     bool flush_telemetry();
00209
00216     bool flush_sensor_data();
00217
00225     bool is_telemetry_collection_time(uint32_t current_time, uint32_t& last_collection_time);
00226
00233     bool is_telemetry_flush_time(uint32_t& collection_counter);
00234
00235
00240     std::string get_last_telemetry_record_csv();
00241
00246     std::string get_last_sensor_record_csv();
00247
00248     static constexpr int TELEMETRY_BUFFER_SIZE = 20;
00249
00250     TelemetryRecord& get_last_telemetry_record() {
00251         size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) %
TELEMETRY_BUFFER_SIZE;
00252         return telemetry_buffer[last_record_index];
00253     }
00254
00255     SensorDataRecord& get_last_sensor_record() {
00256         size_t last_record_index = (telemetry_buffer_write_index + TELEMETRY_BUFFER_SIZE - 1) %
TELEMETRY_BUFFER_SIZE;
00257         return sensor_data_buffer[last_record_index];
00258     }
00259
00260     size_t get_telemetry_buffer_count() const { return telemetry_buffer_count; }
00261     size_t get_telemetry_buffer_write_index() const { return telemetry_buffer_write_index; }

```

```

00262
00263 private:
00264     TelemetryManager(); // Private constructor
00265     ~TelemetryManager() = default;
00266
00270     static constexpr uint32_t DEFAULT_SAMPLE_INTERVAL_MS = 1000;
00271
00275     static constexpr uint32_t DEFAULT_FLUSH_THRESHOLD = 10;
00276
00277     uint32_t sample_interval_ms = DEFAULT_SAMPLE_INTERVAL_MS;
00278
00282     uint32_t flush_threshold = DEFAULT_FLUSH_THRESHOLD;
00283
00287     TelemetryRecord telemetry_buffer[TELEMETRY_BUFFER_SIZE];
00288     size_t telemetry_buffer_count = 0;
00289     size_t telemetry_buffer_write_index = 0;
00290
00294     SensorDataRecord sensor_data_buffer[TELEMETRY_BUFFER_SIZE];
00295
00299     mutex_t telemetry_mutex;
00300 };
00301 #endif // TELEMETRY_MANAGER_H
00302 // End of TelemetryManager group

```

8.93 lib/utils.cpp File Reference

Implementation of utility functions for the Kubisat firmware.

```

#include "utils.h"
#include "pico/multicore.h"
#include "pico/sync.h"
#include <vector>
#include <queue>
#include <string>
#include <array>
#include "system_state_manager.h"

```

Functions

- `std::string get_level_color` ([VerbosityLevel](#) level)
Gets ANSI color code for verbosity level.
- `std::string get_level_prefix` ([VerbosityLevel](#) level)
Gets text prefix for verbosity level.
- `void uart_print` (const `std::string` &msg, [VerbosityLevel](#) level, `uart_inst_t` *uart)
Prints a message to the UART with a timestamp and core number.

Variables

- static `mutex_t` [uart_mutex](#)
Mutex for UART access protection.

8.93.1 Detailed Description

Implementation of utility functions for the Kubisat firmware.

Definition in file [utils.cpp](#).

8.93.2 Function Documentation

8.93.2.1 `get_level_color()`

```
std::string get_level_color (  
    VerboseLevel 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](#).

8.93.2.2 `get_level_prefix()`

```
std::string get_level_prefix (  
    VerboseLevel 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](#).

8.93.2.3 `uart_print()`

```
void uart_print (  
    const std::string & msg,  
    VerboseLevel level,  
    uart_inst_t * uart)
```

Prints a message to the UART with a timestamp and core number.

Prints a message to UART with timestamp and formatting.

Parameters

<i>msg</i>	The message to print.
<i>uart</i>	The UART instance to use for printing.

Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 58 of file [utils.cpp](#).

8.93.3 Variable Documentation

8.93.3.1 uart_mutex

```
mutex_t uart_mutex [static]
```

Mutex for UART access protection.

Definition at line 17 of file [utils.cpp](#).

8.94 utils.cpp

[Go to the documentation of this file.](#)

```
00001 #include "utils.h"
00002 #include "pico/multicore.h"
00003 #include "pico/sync.h"
00004 #include <vector>
00005 #include <queue>
00006 #include <string>
00007 #include <array>
00008 #include "system_state_manager.h"
00009
00014
00015
00017 static mutex_t uart_mutex;
00018
00019
00025 std::string get_level_color(VerbosityLevel level) {
00026     switch (level) {
00027         case VerbosityLevel::ERROR: return ANSI_RED;
00028         case VerbosityLevel::WARNING: return ANSI_YELLOW;
00029         case VerbosityLevel::INFO: return ANSI_GREEN;
00030         case VerbosityLevel::DEBUG: return ANSI_BLUE;
00031         default: return "";
00032     }
00033 }
00034
00035
00041 std::string get_level_prefix(VerbosityLevel level) {
00042     switch (level) {
00043         case VerbosityLevel::ERROR: return "ERROR: ";
00044         case VerbosityLevel::WARNING: return "WARNING: ";
00045         case VerbosityLevel::INFO: return "INFO: ";
00046         case VerbosityLevel::DEBUG: return "DEBUG: ";
00047         default: return "";
00048     }
00049 }
00050
00058 void uart_print(const std::string& msg, VerbosityLevel level, uart_inst_t* uart) {
00059     if (static_cast<int>(level) >
00060         static_cast<int>(SystemStateManager::get_instance().get_uart_verbosity())) {
00061         return;
00062     }
00063     static bool mutex_initiated = false;
00064     if (!mutex_initiated) {
```

```

00065         mutex_init(&uart_mutex);
00066         mutex_inited = true;
00067     }
00068
00069     uint32_t timestamp = to_ms_since_boot(get_absolute_time());
00070     uint core_num = get_core_num();
00071
00072     std::string color = get_level_color(level);
00073     std::string prefix = get_level_prefix(level);
00074     std::string msg_to_send = "[" + std::to_string(timestamp) + "ms] - Core " +
00075                             std::to_string(core_num) + ": " +
00076                             color + prefix + ANSI_RESET + msg + "\r\n";
00077
00078     mutex_enter_blocking(&uart_mutex);
00079     uart_puts(uart, msg_to_send.c_str());
00080     mutex_exit(&uart_mutex);
00081 }

```

8.95 lib/utils.h File Reference

Utility functions and definitions for the Kubisat firmware.

```

#include <stdio.h>
#include <string>
#include "pico/stdlib.h"
#include "hardware/uart.h"
#include "pin_config.h"
#include <vector>

```

Macros

- `#define ANSI_RED "\033[31m"`
ANSI escape codes for terminal color output.
- `#define ANSI_GREEN "\033[32m"`
- `#define ANSI_YELLOW "\033[33m"`
- `#define ANSI_BLUE "\033[34m"`
- `#define ANSI_RESET "\033[0m"`

Enumerations

- enum class `VerbosityLevel` {
`SILENT` = 0 , `ERROR` = 1 , `WARNING` = 2 , `INFO` = 3 ,
`DEBUG` = 4 }
Verbosity levels for logging system.

Functions

- void `uart_print` (const std::string &msg, `VerbosityLevel` level=`VerbosityLevel::INFO`, uart_inst_t *uart=`DEBUG_UART_PORT`)
Prints a message to UART with timestamp and formatting.

8.95.1 Detailed Description

Utility functions and definitions for the Kubisat firmware.

Contains UART logging, color definitions, and CRC calculations

Definition in file [utils.h](#).

8.95.2 Macro Definition Documentation

8.95.2.1 ANSI_RED

```
#define ANSI_RED "\033[31m"
```

ANSI escape codes for terminal color output.

Definition at line 20 of file [utils.h](#).

8.95.2.2 ANSI_GREEN

```
#define ANSI_GREEN "\033[32m"
```

Definition at line 21 of file [utils.h](#).

8.95.2.3 ANSI_YELLOW

```
#define ANSI_YELLOW "\033[33m"
```

Definition at line 22 of file [utils.h](#).

8.95.2.4 ANSI_BLUE

```
#define ANSI_BLUE "\033[34m"
```

Definition at line 23 of file [utils.h](#).

8.95.2.5 ANSI_RESET

```
#define ANSI_RESET "\033[0m"
```

Definition at line 24 of file [utils.h](#).

8.95.3 Enumeration Type Documentation

8.95.3.1 VerbosityLevel

```
enum class VerbosityLevel [strong]
```

Verbosity levels for logging system.

Enumerator

SILENT	No output
ERROR	Only critical errors
WARNING	Warnings and errors
INFO	Normal operation information
DEBUG	Detailed debug information

Definition at line 30 of file [utils.h](#).

8.95.4 Function Documentation

8.95.4.1 uart_print()

```
void uart_print (
    const std::string & msg,
    VerbosityLevel level,
    uart_inst_t * uart)
```

Prints a message to UART with timestamp and formatting.

Parameters

<i>msg</i>	The message to print
<i>level</i>	Message verbosity level
<i>uart</i>	The UART port to use

Prints a message to UART with timestamp and formatting.

Parameters

<i>msg</i>	The message to print.
<i>uart</i>	The UART instance to use for printing.

Prints the given message to the specified UART, prepending it with a timestamp and the core number. Uses a mutex to ensure thread-safe access to the UART.

Definition at line 58 of file [utils.cpp](#).

8.96 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/stdlib.h"
00007 #include "hardware/uart.h"
00008 #include "pin_config.h"
00009 #include <vector>
00010
00011
00017
00018
00020 #define ANSI_RED      "\033[31m"
00021 #define ANSI_GREEN    "\033[32m"
00022 #define ANSI_YELLOW   "\033[33m"
00023 #define ANSI_BLUE     "\033[34m"
00024 #define ANSI_RESET    "\033[0m"
00025
00026
00030 enum class VerbosityLevel {
00031     SILENT = 0,
00032     ERROR = 1,
00033     WARNING = 2,
00034     INFO = 3,
00035     DEBUG = 4
00036 };
00037
00038
00045 void uart_print(const std::string& msg,
00046                 VerbosityLevel level = VerbosityLevel::INFO,
00047                 uart_inst_t* uart = DEBUG_UART_PORT);
00048
00049
00050 #endif
```

8.97 main.cpp File Reference

```
#include "includes.h"
```

Macros

- `#define LOG_FILENAME "/log.txt"`

Functions

- `void core1_entry ()`
- `bool init_pico_hw ()`
- `bool init_modules ()`
- `int main ()`

Variables

- `char buffer [BUFFER_SIZE] = {0}`
- `int buffer_index = 0`

8.97.1 Macro Definition Documentation

8.97.1.1 LOG_FILENAME

```
#define LOG_FILENAME "/log.txt"
```

Definition at line 3 of file [main.cpp](#).

8.97.2 Function Documentation

8.97.2.1 core1_entry()

```
void core1_entry ()
```

Definition at line 8 of file [main.cpp](#).

8.97.2.2 init_pico_hw()

```
bool init_pico_hw ()
```

Definition at line 53 of file [main.cpp](#).

8.97.2.3 init_modules()

```
bool init_modules ()
```

Definition at line 99 of file [main.cpp](#).

8.97.2.4 main()

```
int main (
    void )
```

Definition at line 160 of file [main.cpp](#).

8.97.3 Variable Documentation

8.97.3.1 buffer

```
char buffer[BUFFER_SIZE] = {0}
```

Definition at line 5 of file [main.cpp](#).

8.97.3.2 buffer_index

```
int buffer_index = 0
```

Definition at line 6 of file [main.cpp](#).

8.98 main.cpp

[Go to the documentation of this file.](#)

```
00001 #include "includes.h"
00002
00003 #define LOG_FILENAME "/log.txt"
00004
00005 char buffer[BUFFER_SIZE] = {0};
00006 int buffer_index = 0;
00007
00008 void core1_entry() {
00009     uart_print("Starting core 1", VerbosityLevel::DEBUG);
00010     EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::CORE1_START);
00011
00012     uint32_t last_clock_check_time = 0;
00013     uint32_t last_telemetry_time = 0;
00014     uint32_t telemetry_collection_counter = 0;
00015
00016     TelemetryManager::get_instance().init();
00017
00018     while (true) {
00019         collect_gps_data();
00020
00021         uint32_t currentTime = to_ms_since_boot(get_absolute_time());
00022
00023         uint32_t check_interval_ms = DS3231::get_instance().get_clock_sync_interval() * 60000;
00024         if (currentTime - last_clock_check_time >= check_interval_ms) {
00025             last_clock_check_time = currentTime;
00026
00027             if (DS3231::get_instance().is_sync_needed()) {
00028                 uart_print("Clock sync interval reached, attempting sync", VerbosityLevel::INFO);
00029                 DS3231::get_instance().sync_clock_with_gps();
00030             }
00031         }
00032     }
00033 }
```

```

00030     }
00031 }
00032
00033     if (TelemetryManager::get_instance().is_telemetry_collection_time(currentTime,
last_telemetry_time)) {
00034         TelemetryManager::get_instance().collect_telemetry();
00035         telemetry_collection_counter++;
00036
00037         if
(TelemetryManager::get_instance().is_telemetry_flush_time(telemetry_collection_counter)) {
00038             TelemetryManager::get_instance().flush_telemetry();
00039             telemetry_collection_counter = 0;
00040         }
00041     }
00042
00043     if (SystemStateManager::get_instance().is_bootloader_reset_pending()) {
00044         sleep_ms(100);
00045         uart_print("Entering BOOTSEL mode...", VerbosityLevel::WARNING);
00046         reset_usb_boot(0, 0);
00047     }
00048
00049     sleep_ms(10);
00050 }
00051 }
00052
00053 bool init_pico_hw() {
00054     stdio_init_all();
00055
00056     uart_init(DEBUG_UART_PORT, DEBUG_UART_BAUD_RATE);
00057     gpio_set_function(DEBUG_UART_TX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_TX_PIN));
00058     gpio_set_function(DEBUG_UART_RX_PIN, UART_FUNCSEL_NUM(DEBUG_UART_PORT, DEBUG_UART_RX_PIN));
00059
00060     uart_init(GPS_UART_PORT, GPS_UART_BAUD_RATE);
00061     gpio_set_function(GPS_UART_TX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_TX_PIN));
00062     gpio_set_function(GPS_UART_RX_PIN, UART_FUNCSEL_NUM(GPS_UART_PORT, GPS_UART_RX_PIN));
00063
00064     gpio_init(PICO_DEFAULT_LED_PIN);
00065     gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
00066
00067     gpio_init(PICO_DEFAULT_LED_PIN);
00068     gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
00069     gpio_put(PICO_DEFAULT_LED_PIN, 1);
00070
00071     i2c_init(MAIN_I2C_PORT, 400 * 1000);
00072     gpio_set_function(MAIN_I2C_SCL_PIN, GPIO_FUNC_I2C);
00073     gpio_set_function(MAIN_I2C_SDA_PIN, GPIO_FUNC_I2C);
00074     gpio_pull_up(MAIN_I2C_SCL_PIN);
00075     gpio_pull_up(MAIN_I2C_SDA_PIN);
00076
00077     gpio_init(GPS_POWER_ENABLE_PIN);
00078     gpio_set_dir(GPS_POWER_ENABLE_PIN, GPIO_OUT);
00079     gpio_put(GPS_POWER_ENABLE_PIN, 1);
00080
00081     i2c_init(SENSORS_I2C_PORT, 400 * 1000);
00082     gpio_set_function(SENSORS_I2C_SCL_PIN, GPIO_FUNC_I2C);
00083     gpio_set_function(SENSORS_I2C_SDA_PIN, GPIO_FUNC_I2C);
00084     gpio_pull_up(SENSORS_I2C_SCL_PIN);
00085     gpio_pull_up(SENSORS_I2C_SDA_PIN);
00086     gpio_init(SENSORS_POWER_ENABLE_PIN);
00087     gpio_set_dir(SENSORS_POWER_ENABLE_PIN, GPIO_OUT);
00088     gpio_put(SENSORS_POWER_ENABLE_PIN, 1);
00089
00090     SystemStateManager::get_instance();
00091
00092     EventEmitter::emit(EventGroup::GPS, GPSEvent::POWER_ON);
00093
00094     system("color");
00095
00096     return true;
00097 }
00098
00099 bool init_modules(){
00100     bool radio_init_status = initialize_radio();
00101     SystemStateManager::get_instance().set_radio_init_ok(radio_init_status);
00102
00103     bool sd_init_status = fs_init();
00104     SystemStateManager::get_instance().set_sd_card_mounted(sd_init_status);
00105
00106     if (sd_init_status) {
00107         FILE *fp = fopen(LOG_FILENAME, "w");
00108         if (fp) {
00109             uart_print("Log file opened.", VerbosityLevel::DEBUG);
00110             int bytes_written = fprintf(fp, "System init started.\n");
00111             uart_print("Written " + std::to_string(bytes_written) + " bytes.", VerbosityLevel::DEBUG);
00112             int close_status = fclose(fp);
00113             uart_print("Close file status: " + std::to_string(close_status), VerbosityLevel::DEBUG);
00114

```

```

00115         struct stat file_stat;
00116         if (stat(LOG_FILENAME, &file_stat) == 0) {
00117             size_t file_size = file_stat.st_size;
00118             uart_print("File size: " + std::to_string(file_size) + " bytes",
VerbosityLevel::DEBUG);
00119         } else {
00120             uart_print("Failed to get file size", VerbosityLevel::ERROR);
00121         }
00122
00123         uart_print("File path: /" + std::string(LOG_FILENAME), VerbosityLevel::DEBUG);
00124     } else {
00125         uart_print("Failed to open log file for writing.", VerbosityLevel::ERROR);
00126     }
00127 }
00128
00129 if (sd_init_status) {
00130     uart_print("SD card init: OK", VerbosityLevel::DEBUG);
00131 } else {
00132     uart_print("SD card init: FAILED", VerbosityLevel::ERROR);
00133 }
00134
00135 if (radio_init_status) {
00136     uart_print("Radio init: OK", VerbosityLevel::DEBUG);
00137 } else {
00138     uart_print("Radio init: FAILED", VerbosityLevel::ERROR);
00139 }
00140
00141 Frame boot = frame_build(OperationType::RES, 0, 0, "HELLO");
00142 send_frame_lora(boot);
00143
00144 uart_print("Initializing sensors...", VerbosityLevel::DEBUG);
00145
00146 SensorWrapper& sensor_wrapper = SensorWrapper::get_instance();
00147 bool light_sensor_init = sensor_wrapper.sensor_init(SensorType::LIGHT, SENSORS_I2C_PORT);
00148 SystemStateManager::get_instance().set_light_sensor_init_ok(light_sensor_init);
00149
00150 bool env_sensor_init = sensor_wrapper.sensor_init(SensorType::ENVIRONMENT, SENSORS_I2C_PORT);
00151 SystemStateManager::get_instance().set_env_sensor_init_ok(env_sensor_init);
00152
00153 if (!light_sensor_init || !env_sensor_init) {
00154     uart_print("One or more sensors failed to initialize", VerbosityLevel::WARNING);
00155 }
00156
00157 return sd_init_status && radio_init_status;
00158 }
00159
00160 int main()
00161 {
00162     init_pico_hw();
00163     sleep_ms(100);
00164     init_modules();
00165     EventEmitter::emit(EventGroup::SYSTEM, SystemEvent::BOOT);
00166     sleep_ms(100);
00167     multicore_launch_core1(core1_entry);
00168
00169     gpio_put(PICO_DEFAULT_LED_PIN, 0);
00170
00171     bool power_manager_init_status = PowerManager::get_instance().initialize();
00172     if (power_manager_init_status) {
00173         std::map<std::string, std::string> power_config = {
00174             {"operating_mode", "continuous"},
00175             {"averaging_mode", "16"},
00176         };
00177         PowerManager::get_instance().configure(power_config);
00178     } else {
00179         uart_print("Power manager init error", VerbosityLevel::ERROR);
00180     }
00181
00182     Frame boot = frame_build(OperationType::RES, 0, 0, "START");
00183     send_frame_lora(boot);
00184
00185     std::string boot_string = "System init completed @ " +
std::to_string(to_ms_since_boot(get_absolute_time())) + " ms";
00186     uart_print(boot_string, VerbosityLevel::WARNING);
00187
00188     gpio_put(PICO_DEFAULT_LED_PIN, 1);
00189
00190     while (true)
00191     {
00192         int packet_size = LoRa.parse_packet();
00193         if (packet_size)
00194         {
00195             on_receive(packet_size);
00196         }
00197
00198         handle_uart_input();
00199     }

```



```
00200
00201     return 0;
00202 }
```

8.99 test/comms/commands/test_clock_commands.cpp File Reference

8.100 test_clock_commands.cpp

[Go to the documentation of this file.](#)

```
00001
```

8.101 test/comms/commands/test_diagnostic_commands.cpp File Reference

```
#include "unity.h"
#include "commands.h"
#include "protocol.h"
#include "build_number.h"
```

Functions

- void [test_handle_get_commands_list](#) (void)
- void [test_handle_get_build_version](#) (void)
- void [test_handle_verbosity](#) (void)
- void [test_handle_enter_bootloader_mode](#) (void)

8.101.1 Function Documentation

8.101.1.1 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
    void )
```

Definition at line 6 of file [test_diagnostic_commands.cpp](#).

8.101.1.2 test_handle_get_build_version()

```
void test_handle_get_build_version (
    void )
```

Definition at line 15 of file [test_diagnostic_commands.cpp](#).

8.101.1.3 test_handle_verbosity()

```
void test_handle_verbosity (
    void )
```

Definition at line 25 of file [test_diagnostic_commands.cpp](#).

8.101.1.4 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
    void )
```

Definition at line 40 of file [test_diagnostic_commands.cpp](#).

8.102 test_diagnostic_commands.cpp

[Go to the documentation of this file.](#)

```
00001 #include "unity.h"
00002 #include "commands.h"
00003 #include "protocol.h"
00004 #include "build_number.h"
00005
00006 void test_handle_get_commands_list(void) {
00007     std::vector<Frame> response = handle_get_commands_list("", OperationType::GET);
00008
00009     TEST_ASSERT_TRUE(response.size() > 0);
00010     TEST_ASSERT_EQUAL(OperationType::SEQ, response[0].operationType);
00011     TEST_ASSERT_EQUAL(1, response[0].group);
00012     TEST_ASSERT_EQUAL(0, response[0].command);
00013 }
00014
00015 void test_handle_get_build_version(void) {
00016     std::vector<Frame> response = handle_get_build_version("", OperationType::GET);
00017
00018     TEST_ASSERT_EQUAL(1, response.size());
00019     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00020     TEST_ASSERT_EQUAL(1, response[0].group);
00021     TEST_ASSERT_EQUAL(1, response[0].command);
00022     TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
00023 }
00024
00025 void test_handle_verbosity(void) {
00026     // Test SET operation
00027     std::vector<Frame> response = handle_verbosity("2", OperationType::SET);
00028     TEST_ASSERT_EQUAL(1, response.size());
00029     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00030     TEST_ASSERT_EQUAL(1, response[0].group);
00031     TEST_ASSERT_EQUAL(8, response[0].command);
00032     TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());
00033
00034     // Test GET operation
00035     response = handle_verbosity("", OperationType::GET);
00036     TEST_ASSERT_EQUAL(1, response.size());
00037     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00038 }
00039
00040 void test_handle_enter_bootloader_mode(void) {
00041     std::vector<Frame> response = handle_enter_bootloader_mode("", OperationType::SET);
00042
00043     TEST_ASSERT_EQUAL(1, response.size());
00044     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00045     TEST_ASSERT_EQUAL(1, response[0].group);
00046     TEST_ASSERT_EQUAL(9, response[0].command);
00047 }
```

8.103 test/comms/commands/test_event_commands.cpp File Reference

8.104 test_event_commands.cpp

[Go to the documentation of this file.](#)

00001

8.105 test/comms/commands/test_gps_commands.cpp File Reference

8.106 test_gps_commands.cpp

[Go to the documentation of this file.](#)

00001

8.107 test/comms/commands/test_power_commands.cpp File Reference

8.108 test_power_commands.cpp

[Go to the documentation of this file.](#)

00001

8.109 test/comms/commands/test_sensor_commands.cpp File Reference

8.110 test_sensor_commands.cpp

[Go to the documentation of this file.](#)

00001

8.111 test/comms/commands/test_storage_commands.cpp File Reference

8.112 test_storage_commands.cpp

[Go to the documentation of this file.](#)

00001

8.113 test/comms/commands/test_telemetry_commands.cpp File Reference

8.114 test_telemetry_commands.cpp

[Go to the documentation of this file.](#)

00001

8.115 test/comms/test_comand_handlers.cpp File Reference

```
#include "unity.h"
#include "protocol.h"
#include "communication.h"
#include "commands.h"
```

Functions

- void [send_frame_uart](#) (const [Frame](#) &frame)
- void [send_frame_lora](#) (const [Frame](#) &frame)
- void [setUp](#) (void)
- void [tearDown](#) (void)
- void [test_command_handler_get_operation](#) (void)
- void [test_command_handler_set_operation](#) (void)
- void [test_command_handler_invalid_operation](#) (void)

Variables

- static bool [uart_send_called](#) = false
- static bool [lora_send_called](#) = false
- static [Frame](#) [last_frame_sent](#)

8.115.1 Function Documentation

8.115.1.1 send_frame_uart()

```
void send_frame_uart (
    const Frame & frame)
```

Definition at line 11 of file [test_comand_handlers.cpp](#).

8.115.1.2 send_frame_lora()

```
void send_frame_lora (
    const Frame & frame)
```

Definition at line 16 of file [test_comand_handlers.cpp](#).

8.115.1.3 setUp()

```
void setUp (  
    void )
```

Definition at line 21 of file [test_comand_handlers.cpp](#).

8.115.1.4 tearDown()

```
void tearDown (  
    void )
```

Definition at line 26 of file [test_comand_handlers.cpp](#).

8.115.1.5 test_command_handler_get_operation()

```
void test_command_handler_get_operation (  
    void )
```

Definition at line 29 of file [test_comand_handlers.cpp](#).

8.115.1.6 test_command_handler_set_operation()

```
void test_command_handler_set_operation (  
    void )
```

Definition at line 39 of file [test_comand_handlers.cpp](#).

8.115.1.7 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (  
    void )
```

Definition at line 54 of file [test_comand_handlers.cpp](#).

8.115.2 Variable Documentation

8.115.2.1 uart_send_called

```
bool uart_send_called = false [static]
```

Definition at line 7 of file [test_comand_handlers.cpp](#).

8.115.2.2 lora_send_called

```
bool lora_send_called = false [static]
```

Definition at line 8 of file [test_comand_handlers.cpp](#).

8.115.2.3 last_frame_sent

Frame last_frame_sent [static]

Definition at line 9 of file [test_comand_handlers.cpp](#).

8.116 test_comand_handlers.cpp

[Go to the documentation of this file.](#)

```
00001 // test/comms/test_command_handlers.cpp
00002 #include "unity.h"
00003 #include "protocol.h"
00004 #include "communication.h"
00005 #include "commands.h"
00006
00007 static bool uart_send_called = false;
00008 static bool lora_send_called = false;
00009 static Frame last_frame_sent;
00010
00011 void send_frame_uart(const Frame& frame) {
00012     uart_send_called = true;
00013     last_frame_sent = frame;
00014 }
00015
00016 void send_frame_lora(const Frame& frame) {
00017     lora_send_called = true;
00018     last_frame_sent = frame;
00019 }
00020
00021 void setUp(void) {
00022     uart_send_called = false;
00023     lora_send_called = false;
00024 }
00025
00026 void tearDown(void) {
00027 }
00028
00029 void test_command_handler_get_operation(void) {
00030     std::vector<Frame> response = handle_get_build_version("", OperationType::GET);
00031
00032     TEST_ASSERT_EQUAL(1, response.size());
00033     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00034     TEST_ASSERT_EQUAL(1, response[0].group);
00035     TEST_ASSERT_EQUAL(1, response[0].command);
00036     TEST_ASSERT_EQUAL(BUILD_NUMBER, std::stoi(response[0].value));
00037 }
00038
00039 void test_command_handler_set_operation(void) {
00040     VerbosityLevel old_level = get_verbosity_level();
00041     std::vector<Frame> response = handle_verbosity("2", OperationType::SET);
00042
00043     TEST_ASSERT_EQUAL(1, response.size());
00044     TEST_ASSERT_EQUAL(OperationType::VAL, response[0].operationType);
00045     TEST_ASSERT_EQUAL(1, response[0].group);
00046     TEST_ASSERT_EQUAL(8, response[0].command);
00047     TEST_ASSERT_EQUAL_STRING("LEVEL SET", response[0].value.c_str());
00048
00049     TEST_ASSERT_EQUAL(VerbosityLevel::WARNING, get_verbosity_level());
00050
00051     set_verbosity_level(old_level);
00052 }
00053
00054 void test_command_handler_invalid_operation(void) {
00055     std::vector<Frame> response = handle_get_build_version("", OperationType::SET);
00056
00057     TEST_ASSERT_EQUAL(1, response.size());
00058     TEST_ASSERT_EQUAL(OperationType::ERR, response[0].operationType);
00059     TEST_ASSERT_EQUAL(1, response[0].group);
00060     TEST_ASSERT_EQUAL(1, response[0].command);
00061 }
```

8.117 test/comms/test_converters.cpp File Reference

```
#include "test_frame_common.h"
```

Functions

- void [test_operation_type_conversion](#) ()
- void [test_value_unit_type_conversion](#) ()
- void [test_exception_type_conversion](#) ()
- void [test_hex_string_conversion](#) ()

8.117.1 Function Documentation

8.117.1.1 test_operation_type_conversion()

```
void test_operation_type_conversion (  
    void )
```

Definition at line 4 of file [test_converters.cpp](#).

8.117.1.2 test_value_unit_type_conversion()

```
void test_value_unit_type_conversion (  
    void )
```

Definition at line 13 of file [test_converters.cpp](#).

8.117.1.3 test_exception_type_conversion()

```
void test_exception_type_conversion (  
    void )
```

Definition at line 20 of file [test_converters.cpp](#).

8.117.1.4 test_hex_string_conversion()

```
void test_hex_string_conversion (  
    void )
```

Definition at line 27 of file [test_converters.cpp](#).

8.118 test_converters.cpp

[Go to the documentation of this file.](#)

```

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
00013 void test_value_unit_type_conversion() {
00014     ValueUnit unit = ValueUnit::VOLT;
00015     std::string str = value_unit_type_to_string(unit);
00016
00017     TEST_ASSERT_EQUAL_STRING("V", str.c_str());
00018 }
00019
00020 void test_exception_type_conversion() {
00021     ExceptionType type = ExceptionType::INVALID_PARAM;
00022     std::string str = exception_type_to_string(type);
00023
00024     TEST_ASSERT_EQUAL_STRING("INVALID PARAM", str.c_str());
00025 }
00026
00027 void test_hex_string_conversion() {
00028     std::string hex = "0A0B0C";
00029     std::vector<uint8_t> bytes = hex_string_to_bytes(hex);
00030
00031     TEST_ASSERT_EQUAL(3, bytes.size());
00032     TEST_ASSERT_EQUAL(0x0A, bytes[0]);
00033     TEST_ASSERT_EQUAL(0x0B, bytes[1]);
00034     TEST_ASSERT_EQUAL(0x0C, bytes[2]);
00035 }

```

8.119 test/comms/test_frame_build.cpp File Reference

```
#include "test_frame_common.h"
```

Functions

- void [test_frame_build_val](#) ()
- void [test_frame_build_err](#) ()
- void [test_frame_build_get](#) ()
- void [test_frame_build_set](#) ()
- void [test_frame_build_res](#) ()
- void [test_frame_build_seq](#) ()

8.119.1 Function Documentation

8.119.1.1 test_frame_build_val()

```
void test_frame_build_val (
    void )
```

Definition at line 4 of file [test_frame_build.cpp](#).

8.119.1.2 test_frame_build_err()

```
void test_frame_build_err (  
    void )
```

Definition at line 15 of file [test_frame_build.cpp](#).

8.119.1.3 test_frame_build_get()

```
void test_frame_build_get (  
    void )
```

Definition at line 24 of file [test_frame_build.cpp](#).

8.119.1.4 test_frame_build_set()

```
void test_frame_build_set (  
    void )
```

Definition at line 35 of file [test_frame_build.cpp](#).

8.119.1.5 test_frame_build_res()

```
void test_frame_build_res (  
    void )
```

Definition at line 46 of file [test_frame_build.cpp](#).

8.119.1.6 test_frame_build_seq()

```
void test_frame_build_seq (  
    void )
```

Definition at line 57 of file [test_frame_build.cpp](#).

8.120 test_frame_build.cpp

[Go to the documentation of this file.](#)

```

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 }

```

8.121 test/comms/test_frame_coding.cpp File Reference

```
#include "test_frame_common.h"
```

Functions

- void [test_frame_encode_basic\(\)](#)
- void [test_frame_decode_basic\(\)](#)
- void [test_frame_decode_invalid_header\(\)](#)

8.121.1 Function Documentation

8.121.1.1 test_frame_encode_basic()

```
void test_frame_encode_basic (
    void )
```

Definition at line [4](#) of file [test_frame_coding.cpp](#).

8.121.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (
    void )
```

Definition at line [16](#) of file [test_frame_coding.cpp](#).

8.121.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (
    void )
```

Definition at line [29](#) of file [test_frame_coding.cpp](#).

8.122 test_frame_coding.cpp

[Go to the documentation of this file.](#)

```
00001 // test_frame_codec.cpp
00002 #include "test_frame_common.h"
00003
00004 void test_frame_encode_basic() {
00005     uart_puts(uart0, "start frame_encode_basic");
00006     Frame frame = create_test_frame();
00007     std::string encoded = frame_encode(frame);
00008
00009     TEST_ASSERT_NOT_EQUAL(0, encoded.length());
00010     TEST_ASSERT_TRUE(encoded.find(FRAME_BEGIN) != std::string::npos);
00011     TEST_ASSERT_TRUE(encoded.find(FRAME_END) != std::string::npos);
00012     TEST_ASSERT_TRUE(encoded.find("test_value") != std::string::npos);
00013     uart_puts(uart0, "stop frame_encode_basic");
00014 }
00015
00016 void test_frame_decode_basic() {
00017     Frame original = create_test_frame();
00018     std::string encoded = frame_encode(original);
00019     Frame decoded = frame_decode(encoded);
00020
00021     TEST_ASSERT_EQUAL(original.direction, decoded.direction);
00022     TEST_ASSERT_EQUAL(original.group, decoded.group);
00023     TEST_ASSERT_EQUAL(original.command, decoded.command);
00024     TEST_ASSERT_EQUAL_STRING(original.value.c_str(), decoded.value.c_str());
00025     TEST_ASSERT_EQUAL_STRING(original.unit.c_str(), decoded.unit.c_str());
00026 }
```

```

00027 }
00028
00029 void test_frame_decode_invalid_header() {
00030     std::string invalid_frame = "INVALID" + std::string(1, DELIMITER) + "rest_of_frame";
00031     bool exceptionThrown = false;
00032
00033     try {
00034         Frame decoded = frame_decode(invalid_frame);
00035     } catch (const std::runtime_error& e) {
00036         exceptionThrown = true;
00037     } catch (...) {
00038         // Catch any other exceptions to avoid crashing the test
00039     }
00040
00041     TEST_ASSERT_TRUE(exceptionThrown);
00042 }

```

8.123 test/comms/test_frame_common.h File Reference

```

#include "unity.h"
#include "protocol.h"
#include "communication.h"

```

Functions

- [Frame create_test_frame \(\)](#)

8.123.1 Function Documentation

8.123.1.1 create_test_frame()

```
Frame create_test_frame () [inline]
```

Definition at line 9 of file [test_frame_common.h](#).

8.124 test_frame_common.h

[Go to the documentation of this file.](#)

```

00001 // test_frame_common.h
00002 #ifndef TEST_FRAME_COMMON_H
00003 #define TEST_FRAME_COMMON_H
00004
00005 #include "unity.h"
00006 #include "protocol.h"
00007 #include "communication.h"
00008
00009 inline Frame create_test_frame() {
00010     Frame frame;
00011     frame.header = FRAME_BEGIN;
00012     frame.direction = 1;
00013     frame.operationType = OperationType::GET;
00014     frame.group = 1;
00015     frame.command = 2;
00016     frame.value = "test_value";
00017     frame.unit = "V";
00018     frame.footer = FRAME_END;
00019     return frame;
00020 }
00021
00022 #endif

```

8.125 test/comms/test_frame_send.cpp File Reference

```
#include "unity.h"
#include "communication.h"
#include "../mocks/hardwaremocks.h"
```

Functions

- void [setUp](#) (void)
- void [tearDown](#) (void)
- void [test_send_frame_uart](#) (void)

8.125.1 Function Documentation

8.125.1.1 [setUp\(\)](#)

```
void setUp (
    void )
```

Definition at line 6 of file [test_frame_send.cpp](#).

8.125.1.2 [tearDown\(\)](#)

```
void tearDown (
    void )
```

Definition at line 12 of file [test_frame_send.cpp](#).

8.125.1.3 [test_send_frame_uart\(\)](#)

```
void test_send_frame_uart (
    void )
```

Definition at line 17 of file [test_frame_send.cpp](#).

8.126 test_frame_send.cpp

[Go to the documentation of this file.](#)

```

00001 // test/comms/test_frame_send.cpp
00002 #include "unity.h"
00003 #include "communication.h"
00004 #include "../mocks/hardwaremocks.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 }

```

8.127 test/mocks/hardwaremocks.cpp File Reference

```

#include "hardwaremocks.h"
#include <cstring>

```

Functions

- void [mock_uart_puts](#) (uart_inst_t *uart, const char *str)
- void [mock_uart_init](#) (uart_inst_t *uart, uint baudrate)
- void [mock_spi_write_blocking](#) (spi_inst_t *spi, const uint8_t *src, size_t len)
- int [mock_spi_read_blocking](#) (spi_inst_t *spi, uint8_t tx_data, uint8_t *dst, size_t len)

Variables

- bool [mock_uart_enabled](#) = false
- std::vector< std::string > [uart_output_buffer](#)
- bool [mock_spi_enabled](#) = false
- std::vector< uint8_t > [spi_output_buffer](#)

8.127.1 Function Documentation

8.127.1.1 mock_uart_puts()

```

void mock_uart_puts (
    uart_inst_t * uart,
    const char * str)

```

Definition at line 9 of file [hardwaremocks.cpp](#).

8.127.1.2 mock_uart_init()

```
void mock_uart_init (  
    uart_inst_t * uart,  
    uint baudrate)
```

Definition at line 17 of file [hardware_mocks.cpp](#).

8.127.1.3 mock_spi_write_blocking()

```
void mock_spi_write_blocking (  
    spi_inst_t * spi,  
    const uint8_t * src,  
    size_t len)
```

Definition at line 27 of file [hardware_mocks.cpp](#).

8.127.1.4 mock_spi_read_blocking()

```
int mock_spi_read_blocking (  
    spi_inst_t * spi,  
    uint8_t tx_data,  
    uint8_t * dst,  
    size_t len)
```

Definition at line 35 of file [hardware_mocks.cpp](#).

8.127.2 Variable Documentation

8.127.2.1 mock_uart_enabled

```
bool mock_uart_enabled = false
```

Definition at line 6 of file [hardware_mocks.cpp](#).

8.127.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer
```

Definition at line 7 of file [hardware_mocks.cpp](#).

8.127.2.3 mock_spi_enabled

```
bool mock_spi_enabled = false
```

Definition at line 24 of file [hardware_mocks.cpp](#).

8.127.2.4 spi_output_buffer

`std::vector<uint8_t> spi_output_buffer`

Definition at line 25 of file [hardwaremocks.cpp](#).

8.128 hardwaremocks.cpp

[Go to the documentation of this file.](#)

```

00001 // test/mocks/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 }

```

8.129 test/mocks/hardwaremocks.h File Reference

```

#include <vector>
#include <string>

```

Functions

- void [mock_uart_puts](#) (uart_inst_t *uart, const char *str)
- void [mock_uart_init](#) (uart_inst_t *uart, uint baudrate)
- void [mock_spi_write_blocking](#) (spi_inst_t *spi, const uint8_t *src, size_t len)
- int [mock_spi_read_blocking](#) (spi_inst_t *spi, uint8_t tx_data, uint8_t *dst, size_t len)

Variables

- bool [mock_uart_enabled](#)
- std::vector< std::string > [uart_output_buffer](#)
- bool [mock_spi_enabled](#)
- std::vector< uint8_t > [spi_output_buffer](#)

8.129.1 Function Documentation

8.129.1.1 mock_uart_puts()

```
void mock_uart_puts (  
    uart_inst_t * uart,  
    const char * str)
```

Definition at line 9 of file [hardware_mocks.cpp](#).

8.129.1.2 mock_uart_init()

```
void mock_uart_init (  
    uart_inst_t * uart,  
    uint baudrate)
```

Definition at line 17 of file [hardware_mocks.cpp](#).

8.129.1.3 mock_spi_write_blocking()

```
void mock_spi_write_blocking (  
    spi_inst_t * spi,  
    const uint8_t * src,  
    size_t len)
```

Definition at line 27 of file [hardware_mocks.cpp](#).

8.129.1.4 mock_spi_read_blocking()

```
int mock_spi_read_blocking (  
    spi_inst_t * spi,  
    uint8_t tx_data,  
    uint8_t * dst,  
    size_t len)
```

Definition at line 35 of file [hardware_mocks.cpp](#).

8.129.2 Variable Documentation

8.129.2.1 mock_uart_enabled

```
bool mock_uart_enabled [extern]
```

Definition at line 6 of file [hardware_mocks.cpp](#).

8.129.2.2 uart_output_buffer

```
std::vector<std::string> uart_output_buffer [extern]
```

Definition at line 7 of file [hardware_mock.cpp](#).

8.129.2.3 mock_spi_enabled

```
bool mock_spi_enabled [extern]
```

Definition at line 24 of file [hardware_mock.cpp](#).

8.129.2.4 spi_output_buffer

```
std::vector<uint8_t> spi_output_buffer [extern]
```

Definition at line 25 of file [hardware_mock.cpp](#).

8.130 hardware_mock.h

[Go to the documentation of this file.](#)

```
00001 // test/mocks/hardware_mock.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
```

8.131 test/test_runner.cpp File Reference

```
#include "includes.h"
#include "unity.h"
```

Functions

- void [test_frame_encode_basic](#) (void)
- void [test_frame_decode_basic](#) (void)
- void [test_frame_decode_invalid_header](#) (void)
- void [test_frame_build_get](#) (void)
- void [test_frame_build_set](#) (void)
- void [test_frame_build_res](#) (void)
- void [test_frame_build_seq](#) (void)
- void [test_frame_build_val](#) (void)
- void [test_frame_build_err](#) (void)
- void [test_operation_type_conversion](#) (void)
- void [test_value_unit_type_conversion](#) (void)
- void [test_exception_type_conversion](#) (void)
- void [test_hex_string_conversion](#) (void)
- void [test_command_handler_get_operation](#) (void)
- void [test_command_handler_set_operation](#) (void)
- void [test_command_handler_invalid_operation](#) (void)
- void [test_handle_get_commands_list](#) (void)
- void [test_handle_get_build_version](#) (void)
- void [test_handle_verbosity](#) (void)
- void [test_handle_enter_bootloader_mode](#) (void)
- void [test_error_code_conversion](#) (void)
- int [main](#) (void)

8.131.1 Function Documentation

8.131.1.1 test_frame_encode_basic()

```
void test_frame_encode_basic (  
    void ) [extern]
```

Definition at line 4 of file [test_frame_coding.cpp](#).

8.131.1.2 test_frame_decode_basic()

```
void test_frame_decode_basic (  
    void ) [extern]
```

Definition at line 16 of file [test_frame_coding.cpp](#).

8.131.1.3 test_frame_decode_invalid_header()

```
void test_frame_decode_invalid_header (  
    void ) [extern]
```

Definition at line 29 of file [test_frame_coding.cpp](#).

8.131.1.4 test_frame_build_get()

```
void test_frame_build_get (  
    void ) [extern]
```

Definition at line 24 of file [test_frame_build.cpp](#).

8.131.1.5 test_frame_build_set()

```
void test_frame_build_set (  
    void ) [extern]
```

Definition at line 35 of file [test_frame_build.cpp](#).

8.131.1.6 test_frame_build_res()

```
void test_frame_build_res (  
    void ) [extern]
```

Definition at line 46 of file [test_frame_build.cpp](#).

8.131.1.7 test_frame_build_seq()

```
void test_frame_build_seq (  
    void ) [extern]
```

Definition at line 57 of file [test_frame_build.cpp](#).

8.131.1.8 test_frame_build_val()

```
void test_frame_build_val (  
    void ) [extern]
```

Definition at line 4 of file [test_frame_build.cpp](#).

8.131.1.9 test_frame_build_err()

```
void test_frame_build_err (  
    void ) [extern]
```

Definition at line 15 of file [test_frame_build.cpp](#).

8.131.1.10 test_operation_type_conversion()

```
void test_operation_type_conversion (  
    void ) [extern]
```

Definition at line 4 of file [test_converters.cpp](#).

8.131.1.11 test_value_unit_type_conversion()

```
void test_value_unit_type_conversion (
    void ) [extern]
```

Definition at line 13 of file [test_converters.cpp](#).

8.131.1.12 test_exception_type_conversion()

```
void test_exception_type_conversion (
    void ) [extern]
```

Definition at line 20 of file [test_converters.cpp](#).

8.131.1.13 test_hex_string_conversion()

```
void test_hex_string_conversion (
    void ) [extern]
```

Definition at line 27 of file [test_converters.cpp](#).

8.131.1.14 test_command_handler_get_operation()

```
void test_command_handler_get_operation (
    void ) [extern]
```

Definition at line 29 of file [test_comand_handlers.cpp](#).

8.131.1.15 test_command_handler_set_operation()

```
void test_command_handler_set_operation (
    void ) [extern]
```

Definition at line 39 of file [test_comand_handlers.cpp](#).

8.131.1.16 test_command_handler_invalid_operation()

```
void test_command_handler_invalid_operation (
    void ) [extern]
```

Definition at line 54 of file [test_comand_handlers.cpp](#).

8.131.1.17 test_handle_get_commands_list()

```
void test_handle_get_commands_list (
    void ) [extern]
```

Definition at line 6 of file [test_diagnostic_commands.cpp](#).

8.131.1.18 test_handle_get_build_version()

```
void test_handle_get_build_version (
    void ) [extern]
```

Definition at line 15 of file [test_diagnostic_commands.cpp](#).

8.131.1.19 test_handle_verbosity()

```
void test_handle_verbosity (
    void ) [extern]
```

Definition at line 25 of file [test_diagnostic_commands.cpp](#).

8.131.1.20 test_handle_enter_bootloader_mode()

```
void test_handle_enter_bootloader_mode (
    void ) [extern]
```

Definition at line 40 of file [test_diagnostic_commands.cpp](#).

8.131.1.21 test_error_code_conversion()

```
void test_error_code_conversion (
    void ) [extern]
```

8.131.1.22 main()

```
int main (
    void )
```

Definition at line 38 of file [test_runner.cpp](#).

8.132 test_runner.cpp

[Go to the documentation of this file.](#)

```
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`
 - Constants, [71](#)
 - `_BH1750_DEVICE_ID`
 - Constants, [71](#)
 - `_BH1750_MTREG_MAX`
 - Constants, [71](#)
 - `_BH1750_MTREG_MIN`
 - Constants, [71](#)
 - `__attribute__`
 - Event Management, [55, 56](#)
 - `_filterRes`
 - INA3221, [123](#)
 - `_i2c`
 - INA3221, [123](#)
 - `_i2c_addr`
 - BH1750, [86](#)
 - INA3221, [123](#)
 - `_masken_reg`
 - INA3221, [124](#)
 - `_read`
 - INA3221, [122](#)
 - `_shuntRes`
 - INA3221, [123](#)
 - `_write`
 - INA3221, [123](#)
 - `~ISensor`
 - ISensor, [125](#)
 - `~TelemetryManager`
 - TelemetryManager, [150](#)
- `ADDR_SDO_HIGH`
 - BME280, [92](#)
- `ADDR_SDO_LOW`
 - BME280, [92](#)
- `altitude`
 - TelemetryRecord, [156](#)
- `ANSI_BLUE`
 - utils.h, [273](#)
- `ANSI_GREEN`
 - utils.h, [273](#)
- `ANSI_RED`
 - utils.h, [273](#)
- `ANSI_RESET`
 - utils.h, [273](#)
- `ANSI_YELLOW`
 - utils.h, [273](#)
- `avg_mode`
 - INA3221::conf_reg_t, [107](#)
- `BATTERY_FULL`
 - Event Management, [53](#)
- `BATTERY_FULL_THRESHOLD`
 - PowerManager, [136](#)
- `BATTERY_LOW`
 - Event Management, [53](#)
- `BATTERY_LOW_THRESHOLD`
 - PowerManager, [136](#)
- `BATTERY_NORMAL`
 - Event Management, [53](#)
- `battery_voltage`
 - TelemetryRecord, [154](#)
- `bcd_to_bin`
 - RTC clock, [50](#)
- `begin`
 - BH1750 Light Sensor, [69](#)
 - Configuration Functions, [59](#)
- `BH1750`, [85](#)
 - `_i2c_addr`, [86](#)
 - BH1750 Light Sensor, [69](#)
 - `i2c_port_`, [86](#)
- `BH1750 Light Sensor`, [68](#)
 - `begin`, [69](#)
 - BH1750, [69](#)
 - `configure`, [69](#)
 - `get_light_level`, [70](#)
 - `write8`, [70](#)
- `BH1750Wrapper`, [86](#)
 - BH1750Wrapper, [87](#)
 - `configure`, [89](#)
 - `get_address`, [89](#)
 - `get_i2c_addr`, [88](#)
 - `get_type`, [88](#)
 - `init`, [88](#)
 - `initialized_`, [90](#)
 - `is_initialized`, [88](#)
 - `read_data`, [88](#)
 - `sensor_`, [90](#)
- `bin_to_bcd`
 - RTC clock, [49](#)
- `BME280`, [90](#)
 - `ADDR_SDO_HIGH`, [92](#)
 - `ADDR_SDO_LOW`, [92](#)
 - BME280, [94](#)
 - `calib_params`, [98](#)
 - `configure_sensor`, [97](#)
 - `convert_humidity`, [96](#)
 - `convert_pressure`, [95](#)
 - `convert_temperature`, [95](#)
 - `device_addr`, [98](#)

- get_calibration_parameters, 97
- HUMIDITY_OVERSAMPLING, 94
- i2c_port, 98
- init, 94
- initialized_, 98
- NORMAL_MODE, 94
- NUM_CALIB_PARAMS, 94
- NUM_HUM_CALIB_PARAMS, 94
- OSR_X0, 92
- OSR_X1, 92
- OSR_X16, 92
- OSR_X2, 92
- OSR_X4, 92
- OSR_X8, 92
- Oversampling, 92
- PRESSURE_OVERSAMPLING, 94
- read_raw_all, 95
- read_register, 96, 97
- REG_CONFIG, 93
- REG_CTRL_HUM, 93
- REG_CTRL_MEAS, 93
- REG_DIG_H1, 93
- REG_DIG_H2, 93
- REG_DIG_H3, 93
- REG_DIG_H4, 93
- REG_DIG_H5, 93
- REG_DIG_H6, 93
- REG_DIG_P1_LSB, 93
- REG_DIG_P1_MSB, 93
- REG_DIG_P2_LSB, 93
- REG_DIG_P2_MSB, 93
- REG_DIG_P3_LSB, 93
- REG_DIG_P3_MSB, 93
- REG_DIG_P4_LSB, 93
- REG_DIG_P4_MSB, 93
- REG_DIG_P5_LSB, 93
- REG_DIG_P5_MSB, 93
- REG_DIG_P6_LSB, 93
- REG_DIG_P6_MSB, 93
- REG_DIG_P7_LSB, 93
- REG_DIG_P7_MSB, 93
- REG_DIG_P8_LSB, 93
- REG_DIG_P8_MSB, 93
- REG_DIG_P9_LSB, 93
- REG_DIG_P9_MSB, 93
- REG_DIG_T1_LSB, 93
- REG_DIG_T1_MSB, 93
- REG_DIG_T2_LSB, 93
- REG_DIG_T2_MSB, 93
- REG_DIG_T3_LSB, 93
- REG_DIG_T3_MSB, 93
- REG_HUMIDITY_MSB, 93
- REG_PRESSURE_MSB, 93
- REG_RESET, 93
- REG_TEMPERATURE_MSB, 93
- reset, 94
- t_fine, 98
- TEMPERATURE_OVERSAMPLING, 94
- write_register, 96
- BME280CalibParam, 99
 - dig_h1, 102
 - dig_h2, 102
 - dig_h3, 102
 - dig_h4, 102
 - dig_h5, 102
 - dig_h6, 102
 - dig_p1, 100
 - dig_p2, 100
 - dig_p3, 100
 - dig_p4, 101
 - dig_p5, 101
 - dig_p6, 101
 - dig_p7, 101
 - dig_p8, 101
 - dig_p9, 101
 - dig_t1, 100
 - dig_t2, 100
 - dig_t3, 100
- BME280Wrapper, 103
 - BME280Wrapper, 104
 - configure, 105
 - get_address, 105
 - get_type, 105
 - init, 104
 - initialized_, 106
 - is_initialized, 104
 - read_data, 104
 - sensor_, 106
- BOOL
 - Protocol, 36
- BOOT
 - Event Management, 53
- buffer
 - main.cpp, 276
- buffer_index
 - main.cpp, 276
- BUFFER_SIZE
 - pin_config.h, 225
- BUILD_NUMBER
 - build_number.h, 157
- build_number.h, 157
 - BUILD_NUMBER, 157
- build_version
 - TelemetryRecord, 154
- bus_conv_time
 - INA3221::conf_reg_t, 107
- calib_params
 - BME280, 98
- century
 - ds3231_data_t, 113
- ch1_en
 - INA3221::conf_reg_t, 108
- ch2_en
 - INA3221::conf_reg_t, 108
- ch3_en
 - INA3221::conf_reg_t, 107

- CHANGED
 - Event Management, 54
- charge_current_solar
 - TelemetryRecord, 154
- charge_current_usb
 - TelemetryRecord, 154
- CHARGE_SOLAR
 - power_commands.cpp, 185
- CHARGE_TOTAL
 - power_commands.cpp, 185
- CHARGE_USB
 - power_commands.cpp, 184
- charging_solar_active_
 - PowerManager, 137
- charging_usb_active_
 - PowerManager, 137
- CLOCK
 - Event Management, 53
- Clock Commands, 1
- Clock Management Commands, 11
 - handle_clock_sync_interval, 12
 - handle_get_last_sync_time, 13
 - handle_time, 11
 - handle_timezone_offset, 12
- clock_commands.cpp
 - CLOCK_GROUP, 169
 - CLOCK_SYNC_INTERVAL, 169
 - LAST_SYNC_TIME, 169
 - TIME, 169
 - TIMEZONE_OFFSET, 169
- clock_enable
 - RTC clock, 44
- CLOCK_GROUP
 - clock_commands.cpp, 169
- clock_mutex_
 - DS3231, 111
- CLOCK_SYNC_INTERVAL
 - clock_commands.cpp, 169
- ClockEvent
 - Event Management, 54
- collect_gps_data
 - Location, 57
- collect_gps_telemetry
 - Telemetry Manager, 80
- collect_power_telemetry
 - Telemetry Manager, 79
- collect_sensor_telemetry
 - Telemetry Manager, 80
- collect_telemetry
 - Telemetry Manager, 81
- command
 - Frame, 120
- Command System, 13
 - command_handlers, 15
 - CommandHandler, 14
 - CommandMap, 14
 - execute_command, 14
- command_handlers
 - Command System, 15
- CommandAccessLevel
 - Protocol, 35
- CommandHandler
 - Command System, 14
 - frame.cpp, 201
- CommandMap
 - Command System, 14
- COMMS
 - Event Management, 53
- CommsEvent
 - Event Management, 53
- communication.cpp
 - initialize_radio, 196
 - interval, 197
 - lastPrintTime, 197
 - lastReceiveTime, 197
 - lastSendTime, 197
 - lora_tx_done_callback, 196
 - msgCount, 197
 - outgoing, 197
- communication.h
 - determine_unit, 200
 - initialize_radio, 199
 - lora_tx_done_callback, 199
 - send_frame_lora, 200
 - send_frame_uart, 199
 - send_message, 199
 - split_and_send_message, 200
- Configuration Functions, 58
 - begin, 59
 - get_die_id, 59
 - get_manufacturer_id, 59
 - INA3221, 58
 - read_register, 59
 - reset, 59
 - set_averaging_mode, 61
 - set_bus_conversion_time, 61
 - set_bus_measurement_disable, 61
 - set_bus_measurement_enable, 61
 - set_mode_continuous, 60
 - set_mode_power_down, 60
 - set_mode_triggered, 60
 - set_shunt_conversion_time, 61
 - set_shunt_measurement_disable, 60
 - set_shunt_measurement_enable, 60
- configure
 - BH1750 Light Sensor, 69
 - BH1750Wrapper, 89
 - BME280Wrapper, 105
 - ISensor, 126
 - Power Management, 67
- configure_sensor
 - BME280, 97
- Constants, 70
 - _BH1750_DEFAULT_MTREG, 71
 - _BH1750_DEVICE_ID, 71
 - _BH1750_MTREG_MAX, 71

- [_BH1750_MTREG_MIN](#), 71
- CONTINUOUS_HIGH_RES_MODE
 - Types, 72
- CONTINUOUS_HIGH_RES_MODE_2
 - Types, 72
- CONTINUOUS_LOW_RES_MODE
 - Types, 72
- conv_ready
 - INA3221::masken_reg_t, 127
- convert_humidity
 - BME280, 96
- convert_pressure
 - BME280, 95
- convert_temperature
 - BME280, 95
- core1_entry
 - main.cpp, 275
- CORE1_START
 - Event Management, 53
- CORE1_STOP
 - Event Management, 53
- course
 - TelemetryRecord, 156
- create_test_frame
 - test_frame_common.h, 290
- crit_alert_ch1
 - INA3221::masken_reg_t, 128
- crit_alert_ch2
 - INA3221::masken_reg_t, 128
- crit_alert_ch3
 - INA3221::masken_reg_t, 128
- crit_alert_latch_en
 - INA3221::masken_reg_t, 128
- DATA_READY
 - Event Management, 54
- date
 - ds3231_data_t, 113
 - TelemetryRecord, 156
- DATETIME
 - Protocol, 36
- day
 - ds3231_data_t, 113
- days_of_week
 - DS3231.h, 166
- DEBUG
 - utils.h, 273
- DEBUG_UART_BAUD_RATE
 - pin_config.h, 223
- DEBUG_UART_PORT
 - pin_config.h, 223
- DEBUG_UART_RX_PIN
 - pin_config.h, 224
- DEBUG_UART_TX_PIN
 - pin_config.h, 223
- DEFAULT_FLUSH_THRESHOLD
 - Telemetry Manager, 79
 - TelemetryManager, 152
- DEFAULT_SAMPLE_INTERVAL_MS
 - Telemetry Manager, 78
 - TelemetryManager, 151
- DELIMITER
 - protocol.h, 205
- determine_unit
 - communication.h, 200
- device_addr
 - BME280, 98
- Diagnostic Commands, 16
 - handle_enter_bootloader_mode, 17
 - handle_get_build_version, 16
 - handle_get_commands_list, 16
 - handle_verbosity, 17
- dig_h1
 - BME280CalibParam, 102
- dig_h2
 - BME280CalibParam, 102
- dig_h3
 - BME280CalibParam, 102
- dig_h4
 - BME280CalibParam, 102
- dig_h5
 - BME280CalibParam, 102
- dig_h6
 - BME280CalibParam, 102
- dig_p1
 - BME280CalibParam, 100
- dig_p2
 - BME280CalibParam, 100
- dig_p3
 - BME280CalibParam, 100
- dig_p4
 - BME280CalibParam, 101
- dig_p5
 - BME280CalibParam, 101
- dig_p6
 - BME280CalibParam, 101
- dig_p7
 - BME280CalibParam, 101
- dig_p8
 - BME280CalibParam, 101
- dig_p9
 - BME280CalibParam, 101
- dig_t1
 - BME280CalibParam, 100
- dig_t2
 - BME280CalibParam, 100
- dig_t3
 - BME280CalibParam, 100
- direction
 - Frame, 120
- discharge_current
 - TelemetryRecord, 155
- DRAW_TOTAL
 - power_commands.cpp, 185
- DS3231, 108
 - clock_mutex_, 111
 - DS3231, 110

- ds3231_addr, 111
- i2c, 111
- last_sync_time_, 111
- operator=, 110
- RTC clock, 41
- sync_interval_minutes_, 111
- timezone_offset_minutes_, 111
- DS3231.h
 - days_of_week, 166
 - DS3231_CONTROL_REG, 165
 - DS3231_CONTROL_STATUS_REG, 166
 - DS3231_DATE_REG, 165
 - DS3231_DAY_REG, 165
 - DS3231_DEVICE_ADRESS, 164
 - DS3231_HOURS_REG, 165
 - DS3231_MINUTES_REG, 164
 - DS3231_MONTH_REG, 165
 - DS3231_SECONDS_REG, 164
 - DS3231_TEMPERATURE_LSB_REG, 166
 - DS3231_TEMPERATURE_MSB_REG, 166
 - DS3231_YEAR_REG, 165
 - FRIDAY, 166
 - MONDAY, 166
 - SATURDAY, 166
 - SUNDAY, 166
 - THURSDAY, 166
 - TUESDAY, 166
 - WEDNESDAY, 166
- ds3231_addr
 - DS3231, 111
- DS3231_CONTROL_REG
 - DS3231.h, 165
- DS3231_CONTROL_STATUS_REG
 - DS3231.h, 166
- ds3231_data_t, 112
 - century, 113
 - date, 113
 - day, 113
 - hours, 112
 - minutes, 112
 - month, 113
 - seconds, 112
 - year, 113
- DS3231_DATE_REG
 - DS3231.h, 165
- DS3231_DAY_REG
 - DS3231.h, 165
- DS3231_DEVICE_ADRESS
 - DS3231.h, 164
- DS3231_HOURS_REG
 - DS3231.h, 165
- DS3231_MINUTES_REG
 - DS3231.h, 164
- DS3231_MONTH_REG
 - DS3231.h, 165
- DS3231_SECONDS_REG
 - DS3231.h, 164
- DS3231_TEMPERATURE_LSB_REG
 - DS3231.h, 166
- DS3231.h, 166
- DS3231_TEMPERATURE_MSB_REG
 - DS3231.h, 166
- DS3231_YEAR_REG
 - DS3231.h, 165
- emit
 - EventEmitter, 114
- emit_power_events
 - Telemetry Manager, 80
- env_sensor_init_status
 - SystemStateManager, 148
- ENVIRONMENT
 - Sensors, 73
- ERR
 - Protocol, 35
- ERROR
 - Event Management, 54
 - utils.h, 273
- error_code_to_string
 - Utility Converters, 39
- ErrorCode
 - Protocol, 34
- event
 - event_manager.h, 216
 - EventLog, 115
- Event Commands, 18
 - handle_get_event_count, 19
 - handle_get_last_events, 18
- Event Management, 51
 - __attribute__, 55, 56
 - BATTERY_FULL, 53
 - BATTERY_LOW, 53
 - BATTERY_NORMAL, 53
 - BOOT, 53
 - CHANGED, 54
 - CLOCK, 53
 - ClockEvent, 54
 - COMMS, 53
 - CommsEvent, 53
 - CORE1_START, 53
 - CORE1_STOP, 53
 - DATA_READY, 54
 - ERROR, 54
 - EVENT_BUFFER_SIZE, 52
 - EVENT_FLUSH_THRESHOLD, 52
 - EVENT_LOG_FILE, 52
 - EventGroup, 52
 - get_event, 55
 - GPS, 53
 - GPS_SYNC, 54
 - GPS_SYNC_DATA_NOT_READY, 54
 - GPSEvent, 54
 - init, 55
 - LOCK, 54
 - log_event, 55
 - LOST, 54
 - MSG_RECEIVED, 54
 - MSG_SENT, 54

- PASS_THROUGH_END, [54](#)
- PASS_THROUGH_START, [54](#)
- POWER, [53](#)
- POWER_FALLING, [53](#)
- POWER_OFF, [54](#)
- POWER_ON, [54](#)
- PowerEvent, [53](#)
- RADIO_ERROR, [54](#)
- RADIO_INIT, [54](#)
- save_to_storage, [55](#)
- SHUTDOWN, [53](#)
- SOLAR_ACTIVE, [53](#)
- SOLAR_INACTIVE, [53](#)
- SYSTEM, [53](#)
- SystemEvent, [53](#)
- UART_ERROR, [54](#)
- USB_CONNECTED, [53](#)
- USB_DISCONNECTED, [53](#)
- WATCHDOG_RESET, [53](#)
- EVENT_BUFFER_SIZE
 - Event Management, [52](#)
- EVENT_FLUSH_THRESHOLD
 - Event Management, [52](#)
- EVENT_LOG_FILE
 - Event Management, [52](#)
- event_manager.h
 - event, [216](#)
 - group, [216](#)
 - id, [216](#)
 - timestamp, [216](#)
- eventCount
 - EventManager, [118](#)
- EventEmitter, [114](#)
 - emit, [114](#)
- EventGroup
 - Event Management, [52](#)
- EventLog, [115](#)
 - event, [115](#)
 - group, [115](#)
 - id, [115](#)
 - timestamp, [115](#)
- EventManager, [116](#)
 - eventCount, [118](#)
 - EventManager, [117](#)
 - eventMutex, [118](#)
 - events, [118](#)
 - eventsSinceFlush, [119](#)
 - get_event_count, [117](#)
 - get_instance, [117](#)
 - load_from_storage, [118](#)
 - nextEventId, [118](#)
 - operator=, [117](#)
 - writeIndex, [118](#)
- eventMutex
 - EventManager, [118](#)
- eventRegister
 - frame.cpp, [201](#)
- events
 - EventManager, [118](#)
 - eventsSinceFlush
 - EventManager, [119](#)
 - exception_type_to_string
 - Utility Converters, [38](#)
 - ExceptionType
 - Protocol, [36](#)
 - execute_command
 - Command System, [14](#)
- FAIL_TO_SET
 - Protocol, [35](#)
- fix_quality
 - TelemetryRecord, [156](#)
- flush_sensor_data
 - TelemetryManager, [150](#)
- flush_telemetry
 - Telemetry Manager, [81](#)
- flush_threshold
 - TelemetryManager, [152](#)
- footer
 - Frame, [120](#)
- Frame, [119](#)
 - command, [120](#)
 - direction, [120](#)
 - footer, [120](#)
 - group, [120](#)
 - header, [120](#)
 - operationType, [120](#)
 - unit, [120](#)
 - value, [120](#)
- Frame Handling, [31](#)
 - frame_build, [33](#)
 - frame_decode, [32](#)
 - frame_encode, [32](#)
 - frame_process, [33](#)
- frame.cpp
 - CommandHandler, [201](#)
 - eventRegister, [201](#)
- FRAME_BEGIN
 - protocol.h, [205](#)
- frame_build
 - Frame Handling, [33](#)
- frame_decode
 - Frame Handling, [32](#)
- frame_encode
 - Frame Handling, [32](#)
- FRAME_END
 - protocol.h, [205](#)
- frame_process
 - Frame Handling, [33](#)
- FRIDAY
 - DS3231.h, [166](#)
- fs_init
 - Storage, [76](#)
- fs_stop
 - Storage, [76](#)
- GET

- Protocol, [35](#)
- get_address
 - BH1750Wrapper, [89](#)
 - BME280Wrapper, [105](#)
 - ISensor, [126](#)
- get_available_sensors
 - Sensors, [75](#)
- get_calibration_parameters
 - BME280, [97](#)
- get_clock_sync_interval
 - RTC clock, [45](#)
- get_current
 - INA3221, [123](#)
- get_current_charge_solar
 - Power Management, [66](#)
- get_current_charge_total
 - Power Management, [66](#)
- get_current_charge_usb
 - Power Management, [66](#)
- get_current_draw
 - Power Management, [66](#)
- get_current_ma
 - Measurement Functions, [62](#)
- get_die_id
 - Configuration Functions, [59](#)
- get_event
 - Event Management, [55](#)
- get_event_count
 - EventManager, [117](#)
- get_gga_tokens
 - NMEADData, [132](#)
- get_i2c_addr
 - BH1750Wrapper, [88](#)
- get_instance
 - EventManager, [117](#)
 - NMEADData, [131](#)
 - Power Management, [64](#)
 - RTC clock, [41](#)
 - SensorWrapper, [140](#)
 - SystemStateManager, [143](#)
 - TelemetryManager, [150](#)
- get_last_sensor_record
 - TelemetryManager, [151](#)
- GET_LAST_SENSOR_RECORD_COMMAND
 - telemetry_commands.cpp, [195](#)
- get_last_sensor_record_csv
 - Telemetry Manager, [83](#)
- get_last_sync_time
 - RTC clock, [46](#)
- get_last_telemetry_record
 - TelemetryManager, [151](#)
- GET_LAST_TELEMETRY_RECORD_COMMAND
 - telemetry_commands.cpp, [194](#)
- get_last_telemetry_record_csv
 - Telemetry Manager, [83](#)
- get_level_color
 - utils.cpp, [270](#)
- get_level_prefix
 - utils.cpp, [270](#)
- get_light_level
 - BH1750 Light Sensor, [70](#)
- get_local_time
 - RTC clock, [46](#)
- get_manufacturer_id
 - Configuration Functions, [59](#)
- get_rmc_tokens
 - NMEADData, [132](#)
- get_sensor
 - Sensors, [75](#)
- get_shunt_voltage
 - Measurement Functions, [62](#)
- get_telemetry_buffer_count
 - TelemetryManager, [151](#)
- get_telemetry_buffer_write_index
 - TelemetryManager, [151](#)
- get_time
 - RTC clock, [42](#)
- get_timezone_offset
 - RTC clock, [44](#)
- get_type
 - BH1750Wrapper, [88](#)
 - BME280Wrapper, [105](#)
 - ISensor, [126](#)
- get_uart_verbosity
 - SystemStateManager, [145](#)
- get_unix_time
 - NMEADData, [133](#)
 - RTC clock, [44](#)
- get_voltage
 - Measurement Functions, [63](#)
- get_voltage_5v
 - Power Management, [65](#)
- get_voltage_battery
 - Power Management, [65](#)
- GGA_DATA_COMMAND
 - gps_commands.cpp, [180](#)
- gga_mutex_
 - NMEADData, [133](#)
- gga_tokens_
 - NMEADData, [133](#)
- GPS
 - Event Management, [53](#)
- GPS Commands, [20](#)
 - handle_enable_gps_uart_passthrough, [20](#)
 - handle_get_gga_data, [21](#)
 - handle_get_rmc_data, [21](#)
 - handle_gps_power_status, [20](#)
- gps_collection_paused
 - SystemStateManager, [147](#)
- gps_commands.cpp
 - GGA_DATA_COMMAND, [180](#)
 - GPS_GROUP, [180](#)
 - PASSTHROUGH_COMMAND, [180](#)
 - POWER_STATUS_COMMAND, [180](#)
 - RMC_DATA_COMMAND, [180](#)
- GPS_GROUP

- gps_commands.cpp, 180
- GPS_POWER_ENABLE_PIN
 - pin_config.h, 225
- GPS_SYNC
 - Event Management, 54
- GPS_SYNC_DATA_NOT_READY
 - Event Management, 54
- GPS_UART_BAUD_RATE
 - pin_config.h, 224
- GPS_UART_PORT
 - pin_config.h, 224
- GPS_UART_RX_PIN
 - pin_config.h, 224
- GPS_UART_TX_PIN
 - pin_config.h, 224
- GPSEvent
 - Event Management, 54
- group
 - event_manager.h, 216
 - EventLog, 115
 - Frame, 120
- handle_clock_sync_interval
 - Clock Management Commands, 12
- handle_enable_gps_uart_passthrough
 - GPS Commands, 20
- handle_enter_bootloader_mode
 - Diagnostic Commands, 17
- handle_get_build_version
 - Diagnostic Commands, 16
- handle_get_commands_list
 - Diagnostic Commands, 16
- handle_get_current_charge_solar
 - Power Commands, 25
- handle_get_current_charge_total
 - Power Commands, 25
- handle_get_current_charge_usb
 - Power Commands, 24
- handle_get_current_draw
 - Power Commands, 26
- handle_get_event_count
 - Event Commands, 19
- handle_get_gga_data
 - GPS Commands, 21
- handle_get_last_events
 - Event Commands, 18
- handle_get_last_sensor_record
 - Telemetry Buffer Commands, 31
- handle_get_last_sync_time
 - Clock Management Commands, 13
- handle_get_last_telemetry_record
 - Telemetry Buffer Commands, 30
- handle_get_power_manager_ids
 - Power Commands, 23
- handle_get_rmc_data
 - GPS Commands, 21
- handle_get_sensor_data
 - Sensor Commands, 27
- handle_get_sensor_list
 - Sensor Commands, 28
- handle_get_voltage_5v
 - Power Commands, 24
- handle_get_voltage_battery
 - Power Commands, 23
- handle_gps_power_status
 - GPS Commands, 20
- handle_list_files
 - Storage Commands, 29
- handle_mount
 - Storage Commands, 29
- handle_sensor_config
 - Sensor Commands, 27
- handle_time
 - Clock Management Commands, 11
- handle_timezone_offset
 - Clock Management Commands, 12
- handle_uart_input
 - Receiving Data, 37
- handle_verbosity
 - Diagnostic Commands, 17
- hardware_mock.cpp
 - mock_spi_enabled, 293
 - mock_spi_read_blocking, 293
 - mock_spi_write_blocking, 293
 - mock_uart_enabled, 293
 - mock_uart_init, 292
 - mock_uart_puts, 292
 - spi_output_buffer, 293
 - uart_output_buffer, 293
- hardware_mock.h
 - mock_spi_enabled, 296
 - mock_spi_read_blocking, 295
 - mock_spi_write_blocking, 295
 - mock_uart_enabled, 295
 - mock_uart_init, 295
 - mock_uart_puts, 295
 - spi_output_buffer, 296
 - uart_output_buffer, 295
- has_valid_time
 - NMEAData, 132
- header
 - Frame, 120
- hex_string_to_bytes
 - Utility Converters, 39
- hours
 - ds3231_data_t, 112
- HUMIDITY
 - Sensors, 74
- humidity
 - SensorDataRecord, 139
- HUMIDITY_OVERSAMPLING
 - BME280, 94
- i2c
 - DS3231, 111
- i2c_port
 - BME280, 98
- i2c_port_

- BH1750, [86](#)
- i2c_read_reg
 - RTC clock, [48](#)
- i2c_write_reg
 - RTC clock, [48](#)
- id
 - event_manager.h, [216](#)
 - EventLog, [115](#)
- INA3221, [121](#)
 - _filterRes, [123](#)
 - _i2c, [123](#)
 - _i2c_addr, [123](#)
 - _masken_reg, [124](#)
 - _read, [122](#)
 - _shuntRes, [123](#)
 - _write, [123](#)
 - Configuration Functions, [58](#)
 - get_current, [123](#)
- INA3221 Power Monitor, [57](#)
- INA3221.h
 - INA3221_ADDR40_GND, [234](#)
 - INA3221_ADDR41_VCC, [234](#)
 - INA3221_ADDR42_SDA, [234](#)
 - INA3221_ADDR43_SCL, [234](#)
 - ina3221_addr_t, [234](#)
 - ina3221_avg_mode_t, [237](#)
 - INA3221_CH1, [236](#)
 - INA3221_CH2, [236](#)
 - INA3221_CH3, [236](#)
 - INA3221_CH_NUM, [237](#)
 - ina3221_ch_t, [234](#)
 - ina3221_conv_time_t, [236](#)
 - INA3221_REG_CH1_BUSV, [236](#)
 - INA3221_REG_CH1_CRIT_ALERT_LIM, [236](#)
 - INA3221_REG_CH1_SHUNTV, [236](#)
 - INA3221_REG_CH1_WARNING_ALERT_LIM, [236](#)
 - INA3221_REG_CH2_BUSV, [236](#)
 - INA3221_REG_CH2_CRIT_ALERT_LIM, [236](#)
 - INA3221_REG_CH2_SHUNTV, [236](#)
 - INA3221_REG_CH2_WARNING_ALERT_LIM, [236](#)
 - INA3221_REG_CH3_BUSV, [236](#)
 - INA3221_REG_CH3_CRIT_ALERT_LIM, [236](#)
 - INA3221_REG_CH3_SHUNTV, [236](#)
 - INA3221_REG_CH3_WARNING_ALERT_LIM, [236](#)
 - INA3221_REG_CONF, [236](#)
 - INA3221_REG_CONF_AVG_1, [237](#)
 - INA3221_REG_CONF_AVG_1024, [237](#)
 - INA3221_REG_CONF_AVG_128, [237](#)
 - INA3221_REG_CONF_AVG_16, [237](#)
 - INA3221_REG_CONF_AVG_256, [237](#)
 - INA3221_REG_CONF_AVG_4, [237](#)
 - INA3221_REG_CONF_AVG_512, [237](#)
 - INA3221_REG_CONF_AVG_64, [237](#)
 - INA3221_REG_CONF_CT_1100US, [237](#)
 - INA3221_REG_CONF_CT_140US, [237](#)
 - INA3221_REG_CONF_CT_204US, [237](#)
 - INA3221_REG_CONF_CT_2116US, [237](#)
 - INA3221_REG_CONF_CT_332US, [237](#)
 - INA3221_REG_CONF_CT_4156US, [237](#)
 - INA3221_REG_CONF_CT_588US, [237](#)
 - INA3221_REG_CONF_CT_8244US, [237](#)
 - INA3221_REG_DIE_ID, [236](#)
 - INA3221_REG_MANUF_ID, [236](#)
 - INA3221_REG_MASK_ENABLE, [236](#)
 - INA3221_REG_PWR_VALID_HI_LIM, [236](#)
 - INA3221_REG_PWR_VALID_LO_LIM, [236](#)
 - INA3221_REG_SHUNTV_SUM, [236](#)
 - INA3221_REG_SHUNTV_SUM_LIM, [236](#)
 - ina3221_reg_t, [236](#)
 - SHUNT_VOLTAGE_LSB_UV, [237](#)
- INA3221::conf_reg_t, [106](#)
 - avg_mode, [107](#)
 - bus_conv_time, [107](#)
 - ch1_en, [108](#)
 - ch2_en, [108](#)
 - ch3_en, [107](#)
 - mode_bus_en, [107](#)
 - mode_continuous_en, [107](#)
 - mode_shunt_en, [107](#)
 - reset, [108](#)
 - shunt_conv_time, [107](#)
- INA3221::masken_reg_t, [127](#)
 - conv_ready, [127](#)
 - crit_alert_ch1, [128](#)
 - crit_alert_ch2, [128](#)
 - crit_alert_ch3, [128](#)
 - crit_alert_latch_en, [128](#)
 - pwr_valid_alert, [127](#)
 - reserved, [129](#)
 - shunt_sum_alert, [128](#)
 - shunt_sum_en_ch1, [129](#)
 - shunt_sum_en_ch2, [129](#)
 - shunt_sum_en_ch3, [129](#)
 - timing_ctrl_alert, [127](#)
 - warn_alert_ch1, [128](#)
 - warn_alert_ch2, [128](#)
 - warn_alert_ch3, [127](#)
 - warn_alert_latch_en, [128](#)
- ina3221_
 - PowerManager, [137](#)
- INA3221_ADDR40_GND
 - INA3221.h, [234](#)
- INA3221_ADDR41_VCC
 - INA3221.h, [234](#)
- INA3221_ADDR42_SDA
 - INA3221.h, [234](#)
- INA3221_ADDR43_SCL
 - INA3221.h, [234](#)
- ina3221_addr_t
 - INA3221.h, [234](#)
- ina3221_avg_mode_t
 - INA3221.h, [237](#)
- INA3221_CH1

INA3221.h, [236](#)
 INA3221_CH2
 INA3221.h, [236](#)
 INA3221_CH3
 INA3221.h, [236](#)
 INA3221_CH_NUM
 INA3221.h, [237](#)
 ina3221_ch_t
 INA3221.h, [234](#)
 ina3221_conv_time_t
 INA3221.h, [236](#)
 INA3221_REG_CH1_BUSV
 INA3221.h, [236](#)
 INA3221_REG_CH1_CRIT_ALERT_LIM
 INA3221.h, [236](#)
 INA3221_REG_CH1_SHUNTV
 INA3221.h, [236](#)
 INA3221_REG_CH1_WARNING_ALERT_LIM
 INA3221.h, [236](#)
 INA3221_REG_CH2_BUSV
 INA3221.h, [236](#)
 INA3221_REG_CH2_CRIT_ALERT_LIM
 INA3221.h, [236](#)
 INA3221_REG_CH2_SHUNTV
 INA3221.h, [236](#)
 INA3221_REG_CH2_WARNING_ALERT_LIM
 INA3221.h, [236](#)
 INA3221_REG_CH3_BUSV
 INA3221.h, [236](#)
 INA3221_REG_CH3_CRIT_ALERT_LIM
 INA3221.h, [236](#)
 INA3221_REG_CH3_SHUNTV
 INA3221.h, [236](#)
 INA3221_REG_CH3_WARNING_ALERT_LIM
 INA3221.h, [236](#)
 INA3221_REG_CONF
 INA3221.h, [236](#)
 INA3221_REG_CONF_AVG_1
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_1024
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_128
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_16
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_256
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_4
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_512
 INA3221.h, [237](#)
 INA3221_REG_CONF_AVG_64
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_1100US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_140US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_204US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_2116US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_332US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_4156US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_588US
 INA3221.h, [237](#)
 INA3221_REG_CONF_CT_8244US
 INA3221.h, [237](#)
 INA3221_REG_DIE_ID
 INA3221.h, [236](#)
 INA3221_REG_MANUF_ID
 INA3221.h, [236](#)
 INA3221_REG_MASK_ENABLE
 INA3221.h, [236](#)
 INA3221_REG_PWR_VALID_HI_LIM
 INA3221.h, [236](#)
 INA3221_REG_PWR_VALID_LO_LIM
 INA3221.h, [236](#)
 INA3221_REG_SHUNTV_SUM
 INA3221.h, [236](#)
 INA3221_REG_SHUNTV_SUM_LIM
 INA3221.h, [236](#)
 ina3221_reg_t
 INA3221.h, [236](#)
 includes.h, [158](#)
 INFO
 utils.h, [273](#)
 init
 BH1750Wrapper, [88](#)
 BME280, [94](#)
 BME280Wrapper, [104](#)
 Event Management, [55](#)
 ISensor, [125](#)
 Telemetry Manager, [81](#)
 init_modules
 main.cpp, [275](#)
 init_pico_hw
 main.cpp, [275](#)
 initialize
 Power Management, [65](#)
 initialize_radio
 communication.cpp, [196](#)
 communication.h, [199](#)
 initialized_
 BH1750Wrapper, [90](#)
 BME280, [98](#)
 BME280Wrapper, [106](#)
 PowerManager, [137](#)
 Interface
 Protocol, [36](#)
 INTERNAL_FAIL_TO_READ
 Protocol, [35](#)
 interval
 communication.cpp, [197](#)
 INVALID_FORMAT

- Protocol, [35](#)
- INVALID_OPERATION
 - Protocol, [35](#), [36](#)
- INVALID_PARAM
 - Protocol, [36](#)
- INVALID_VALUE
 - Protocol, [35](#)
- is_bootloader_reset_pending
 - SystemStateManager, [143](#)
- is_charging_solar
 - Power Management, [68](#)
- is_charging_usb
 - Power Management, [68](#)
- is_env_sensor_init_ok
 - SystemStateManager, [146](#)
- is_gps_collection_paused
 - SystemStateManager, [144](#)
- is_initialized
 - BH1750Wrapper, [88](#)
 - BME280Wrapper, [104](#)
 - ISensor, [125](#)
- is_light_sensor_init_ok
 - SystemStateManager, [146](#)
- is_radio_init_ok
 - SystemStateManager, [145](#)
- is_sd_card_mounted
 - SystemStateManager, [144](#)
- is_sync_needed
 - RTC clock, [47](#)
- is_telemetry_collection_time
 - Telemetry Manager, [82](#)
- is_telemetry_flush_time
 - Telemetry Manager, [82](#)
- ISensor, [124](#)
 - ~ISensor, [125](#)
 - configure, [126](#)
 - get_address, [126](#)
 - get_type, [126](#)
 - init, [125](#)
 - is_initialized, [125](#)
 - read_data, [125](#)
- last_frame_sent
 - test_comand_handlers.cpp, [283](#)
- LAST_SYNC_TIME
 - clock_commands.cpp, [169](#)
- last_sync_time_
 - DS3231, [111](#)
- lastPrintTime
 - communication.cpp, [197](#)
- lastReceiveTime
 - communication.cpp, [197](#)
- lastSendTime
 - communication.cpp, [197](#)
- lat_dir
 - TelemetryRecord, [155](#)
- latitude
 - TelemetryRecord, [155](#)
- lib/clock/DS3231.cpp, [159](#)
- lib/clock/DS3231.h, [163](#), [167](#)
- lib/comms/commands/clock_commands.cpp, [168](#), [170](#)
- lib/comms/commands/commands.cpp, [172](#), [173](#)
- lib/comms/commands/commands.h, [174](#), [175](#)
- lib/comms/commands/diagnostic_commands.cpp, [176](#), [177](#)
- lib/comms/commands/event_commands.cpp, [178](#)
- lib/comms/commands/gps_commands.cpp, [179](#), [181](#)
- lib/comms/commands/power_commands.cpp, [183](#), [185](#)
- lib/comms/commands/sensor_commands.cpp, [187](#), [188](#)
- lib/comms/commands/storage_commands.cpp, [191](#), [192](#)
- lib/comms/commands/telemetry_commands.cpp, [194](#), [195](#)
- lib/comms/communication.cpp, [196](#), [198](#)
- lib/comms/communication.h, [198](#), [200](#)
- lib/comms/frame.cpp, [201](#), [202](#)
- lib/comms/protocol.h, [203](#), [205](#)
- lib/comms/receive.cpp, [207](#), [208](#)
- lib/comms/send.cpp, [209](#), [210](#)
- lib/comms/utils_converters.cpp, [210](#), [211](#)
- lib/eventman/event_manager.cpp, [212](#), [213](#)
- lib/eventman/event_manager.h, [214](#), [217](#)
- lib/location/gps_collector.cpp, [218](#), [219](#)
- lib/location/gps_collector.h, [220](#)
- lib/location/NMEA/NMEA_data.h, [221](#)
- lib/pin_config.h, [222](#), [229](#)
- lib/powerman/INA3221/INA3221.cpp, [229](#), [230](#)
- lib/powerman/INA3221/INA3221.h, [233](#), [238](#)
- lib/powerman/PowerManager.cpp, [240](#)
- lib/powerman/PowerManager.h, [242](#), [243](#)
- lib/sensors/BH1750/BH1750.cpp, [243](#), [244](#)
- lib/sensors/BH1750/BH1750.h, [244](#), [245](#)
- lib/sensors/BH1750/BH1750_WRAPPER.cpp, [246](#)
- lib/sensors/BH1750/BH1750_WRAPPER.h, [247](#)
- lib/sensors/BME280/BME280.cpp, [248](#)
- lib/sensors/BME280/BME280.h, [251](#)
- lib/sensors/BME280/BME280_WRAPPER.cpp, [253](#)
- lib/sensors/BME280/BME280_WRAPPER.h, [254](#)
- lib/sensors/ISensor.cpp, [254](#), [255](#)
- lib/sensors/ISensor.h, [255](#), [256](#)
- lib/storage/storage.cpp, [257](#), [258](#)
- lib/storage/storage.h, [258](#), [259](#)
- lib/system_state_manager.h, [259](#), [260](#)
- lib/telemetry/telemetry_manager.cpp, [261](#), [262](#)
- lib/telemetry/telemetry_manager.h, [266](#), [267](#)
- lib/utils.cpp, [269](#), [271](#)
- lib/utils.h, [272](#), [274](#)
- LIGHT
 - Sensors, [73](#)
- light
 - SensorDataRecord, [139](#)
- LIGHT_LEVEL
 - Sensors, [74](#)
- light_sensor_init_status
 - SystemStateManager, [148](#)
- LIST_FILES_COMMAND
 - storage_commands.cpp, [192](#)

- load_from_storage
 - EventManager, 118
- Location, 56
 - collect_gps_data, 57
 - MAX_RAW_DATA_LENGTH, 57
 - splitString, 57
- LOCK
 - Event Management, 54
- log_event
 - Event Management, 55
- LOG_FILENAME
 - main.cpp, 275
- lon_dir
 - TelemetryRecord, 155
- longitude
 - TelemetryRecord, 155
- LORA
 - Protocol, 36
- lora_address_local
 - pin_config.h, 228
- lora_address_remote
 - pin_config.h, 228
- lora_cs_pin
 - pin_config.h, 228
- LORA_DEFAULT_DIO0_PIN
 - pin_config.h, 227
- LORA_DEFAULT_RESET_PIN
 - pin_config.h, 227
- LORA_DEFAULT_SPI
 - pin_config.h, 227
- LORA_DEFAULT_SPI_FREQUENCY
 - pin_config.h, 227
- LORA_DEFAULT_SS_PIN
 - pin_config.h, 227
- lora_irq_pin
 - pin_config.h, 228
- lora_reset_pin
 - pin_config.h, 228
- lora_send_called
 - test_comand_handlers.cpp, 283
- lora_tx_done_callback
 - communication.cpp, 196
 - communication.h, 199
- LOST
 - Event Management, 54
- main
 - main.cpp, 276
 - test_runner.cpp, 300
- main.cpp, 275
 - buffer, 276
 - buffer_index, 276
 - core1_entry, 275
 - init_modules, 275
 - init_pico_hw, 275
 - LOG_FILENAME, 275
 - main, 276
- MAIN_I2C_PORT
 - pin_config.h, 224
- MAIN_I2C_SCL_PIN
 - pin_config.h, 224
- MAIN_I2C_SDA_PIN
 - pin_config.h, 224
- MAX_PACKET_SIZE
 - receive.cpp, 207
- MAX_RAW_DATA_LENGTH
 - Location, 57
- Measurement Functions, 62
 - get_current_ma, 62
 - get_shunt_voltage, 62
 - get_voltage, 63
- MILIAMP
 - Protocol, 36
- minutes
 - ds3231_data_t, 112
- mock_spi_enabled
 - hardware_mock.cpp, 293
 - hardware_mock.h, 296
- mock_spi_read_blocking
 - hardware_mock.cpp, 293
 - hardware_mock.h, 295
- mock_spi_write_blocking
 - hardware_mock.cpp, 293
 - hardware_mock.h, 295
- mock_uart_enabled
 - hardware_mock.cpp, 293
 - hardware_mock.h, 295
- mock_uart_init
 - hardware_mock.cpp, 292
 - hardware_mock.h, 295
- mock_uart_puts
 - hardware_mock.cpp, 292
 - hardware_mock.h, 295
- Mode
 - Types, 72
- mode_bus_en
 - INA3221::conf_reg_t, 107
- mode_continuous_en
 - INA3221::conf_reg_t, 107
- mode_shunt_en
 - INA3221::conf_reg_t, 107
- MONDAY
 - DS3231.h, 166
- month
 - ds3231_data_t, 113
- MOUNT_COMMAND
 - storage_commands.cpp, 192
- MSG_RECEIVED
 - Event Management, 54
- MSG_SENT
 - Event Management, 54
- msgCount
 - communication.cpp, 197
- mutex_
 - SystemStateManager, 148
- nextEventId
 - EventManager, 118

- NMEADData, 129
 - get_gga_tokens, 132
 - get_instance, 131
 - get_rmc_tokens, 132
 - get_unix_time, 133
 - gga_mutex_, 133
 - gga_tokens_, 133
 - has_valid_time, 132
 - NMEADData, 131
 - operator=, 131
 - rmc_mutex_, 133
 - rmc_tokens_, 133
 - update_gga_tokens, 132
 - update_rmc_tokens, 131
- NONE
 - Protocol, 35, 36
 - Sensors, 73, 74
- NORMAL_MODE
 - BME280, 94
- NOT_ALLOWED
 - Protocol, 35, 36
- NUM_CALIB_PARAMS
 - BME280, 94
- NUM_HUM_CALIB_PARAMS
 - BME280, 94
- on_receive
 - Receiving Data, 37
- ONE_TIME_HIGH_RES_MODE
 - Types, 72
- ONE_TIME_HIGH_RES_MODE_2
 - Types, 72
- ONE_TIME_LOW_RES_MODE
 - Types, 72
- operation_type_to_string
 - Utility Converters, 38
- OperationType
 - Protocol, 35
- operationType
 - Frame, 120
- operator=
 - DS3231, 110
 - EventManager, 117
 - NMEADData, 131
 - PowerManager, 136
 - SystemStateManager, 143
- OSR_X0
 - BME280, 92
- OSR_X1
 - BME280, 92
- OSR_X16
 - BME280, 92
- OSR_X2
 - BME280, 92
- OSR_X4
 - BME280, 92
- OSR_X8
 - BME280, 92
- outgoing
 - communication.cpp, 197
- Oversampling
 - BME280, 92
- PA_OUTPUT_PA_BOOST_PIN
 - pin_config.h, 228
- PA_OUTPUT_RFO_PIN
 - pin_config.h, 227
- PARAM_INVALID
 - Protocol, 35
- PARAM_REQUIRED
 - Protocol, 35
- PARAM_UNECESSARY
 - Protocol, 36
- PARAM_UNNECESSARY
 - Protocol, 35
- PASS_THROUGH_END
 - Event Management, 54
- PASS_THROUGH_START
 - Event Management, 54
- PASSTHROUGH_COMMAND
 - gps_commands.cpp, 180
- pending_bootloader_reset
 - SystemStateManager, 147
- pin_config.h
 - BUFFER_SIZE, 225
 - DEBUG_UART_BAUD_RATE, 223
 - DEBUG_UART_PORT, 223
 - DEBUG_UART_RX_PIN, 224
 - DEBUG_UART_TX_PIN, 223
 - GPS_POWER_ENABLE_PIN, 225
 - GPS_UART_BAUD_RATE, 224
 - GPS_UART_PORT, 224
 - GPS_UART_RX_PIN, 224
 - GPS_UART_TX_PIN, 224
 - lora_address_local, 228
 - lora_address_remote, 228
 - lora_cs_pin, 228
 - LORA_DEFAULT_DIO0_PIN, 227
 - LORA_DEFAULT_RESET_PIN, 227
 - LORA_DEFAULT_SPI, 227
 - LORA_DEFAULT_SPI_FREQUENCY, 227
 - LORA_DEFAULT_SS_PIN, 227
 - lora_irq_pin, 228
 - lora_reset_pin, 228
 - MAIN_I2C_PORT, 224
 - MAIN_I2C_SCL_PIN, 224
 - MAIN_I2C_SDA_PIN, 224
 - PA_OUTPUT_PA_BOOST_PIN, 228
 - PA_OUTPUT_RFO_PIN, 227
 - READ_BIT, 227
 - SD_CARD_DETECT_PIN, 226
 - SD_CS_PIN, 226
 - SD_MISO_PIN, 225
 - SD_MOSI_PIN, 226
 - SD_SCK_PIN, 226
 - SD_SPI_PORT, 225
 - SENSORS_I2C_PORT, 225
 - SENSORS_I2C_SCL_PIN, 225

- SENSORS_I2C_SDA_PIN, 225
- SENSORS_POWER_ENABLE_PIN, 225
- SPI_PORT, 227
- SX1278_CS, 226
- SX1278_MISO, 226
- SX1278_MOSI, 226
- SX1278_SCK, 226
- POWER
 - Event Management, 53
- Power Commands, 22
 - handle_get_current_charge_solar, 25
 - handle_get_current_charge_total, 25
 - handle_get_current_charge_usb, 24
 - handle_get_current_draw, 26
 - handle_get_power_manager_ids, 23
 - handle_get_voltage_5v, 24
 - handle_get_voltage_battery, 23
- Power Management, 63
 - configure, 67
 - get_current_charge_solar, 66
 - get_current_charge_total, 66
 - get_current_charge_usb, 66
 - get_current_draw, 66
 - get_instance, 64
 - get_voltage_5v, 65
 - get_voltage_battery, 65
 - initialize, 65
 - is_charging_solar, 68
 - is_charging_usb, 68
 - PowerManager, 64
 - read_device_ids, 65
- power_commands.cpp
 - CHARGE_SOLAR, 185
 - CHARGE_TOTAL, 185
 - CHARGE_USB, 184
 - DRAW_TOTAL, 185
 - POWER_GROUP, 184
 - POWER_MANAGER_IDS, 184
 - VOLTAGE_BATTERY, 184
 - VOLTAGE_MAIN, 184
- POWER_FALLING
 - Event Management, 53
- POWER_GROUP
 - power_commands.cpp, 184
- POWER_MANAGER_IDS
 - power_commands.cpp, 184
- POWER_OFF
 - Event Management, 54
- POWER_ON
 - Event Management, 54
 - Types, 72
- POWER_STATUS_COMMAND
 - gps_commands.cpp, 180
- PowerEvent
 - Event Management, 53
- powerman_mutex_
 - PowerManager, 137
- PowerManager, 134
 - BATTERY_FULL_THRESHOLD, 136
 - BATTERY_LOW_THRESHOLD, 136
 - charging_solar_active_, 137
 - charging_usb_active_, 137
 - ina3221_, 137
 - initialized_, 137
 - operator=, 136
 - Power Management, 64
 - powerman_mutex_, 137
 - PowerManager, 135, 136
 - SOLAR_CURRENT_THRESHOLD, 136
 - USB_CURRENT_THRESHOLD, 136
- PRESSURE
 - Sensors, 74
- pressure
 - SensorDataRecord, 138
- PRESSURE_OVERSAMPLING
 - BME280, 94
- Protocol, 34
 - BOOL, 36
 - CommandAccessLevel, 35
 - DATETIME, 36
 - ERR, 35
 - ErrorCode, 34
 - ExceptionType, 36
 - FAIL_TO_SET, 35
 - GET, 35
 - Interface, 36
 - INTERNAL_FAIL_TO_READ, 35
 - INVALID_FORMAT, 35
 - INVALID_OPERATION, 35, 36
 - INVALID_PARAM, 36
 - INVALID_VALUE, 35
 - LORA, 36
 - MILIAMP, 36
 - NONE, 35, 36
 - NOT_ALLOWED, 35, 36
 - OperationType, 35
 - PARAM_INVALID, 35
 - PARAM_REQUIRED, 35
 - PARAM_UNNECESSARY, 36
 - PARAM_UNNECESSARY, 35
 - READ_ONLY, 35
 - READ_WRITE, 35
 - RES, 35
 - SECOND, 36
 - SEQ, 35
 - SET, 35
 - TEXT, 36
 - UART, 36
 - UNDEFINED, 36
 - UNKNOWN_ERROR, 35
 - VAL, 35
 - ValueUnit, 35
 - VOLT, 36
 - WRITE_ONLY, 35
- protocol.h
 - DELIMITER, 205

- FRAME_BEGIN, [205](#)
- FRAME_END, [205](#)
- pwr_valid_alert
 - INA3221::masken_reg_t, [127](#)
- RADIO_ERROR
 - Event Management, [54](#)
- RADIO_INIT
 - Event Management, [54](#)
- radio_init_status
 - SystemStateManager, [147](#)
- READ_BIT
 - pin_config.h, [227](#)
- read_data
 - BH1750Wrapper, [88](#)
 - BME280Wrapper, [104](#)
 - ISensor, [125](#)
- read_device_ids
 - Power Management, [65](#)
- READ_ONLY
 - Protocol, [35](#)
- read_raw_all
 - BME280, [95](#)
- read_register
 - BME280, [96](#), [97](#)
 - Configuration Functions, [59](#)
- read_temperature
 - RTC clock, [42](#)
- READ_WRITE
 - Protocol, [35](#)
- receive.cpp
 - MAX_PACKET_SIZE, [207](#)
- Receiving Data, [36](#)
 - handle_uart_input, [37](#)
 - on_receive, [37](#)
- REG_CONFIG
 - BME280, [93](#)
- REG_CTRL_HUM
 - BME280, [93](#)
- REG_CTRL_MEAS
 - BME280, [93](#)
- REG_DIG_H1
 - BME280, [93](#)
- REG_DIG_H2
 - BME280, [93](#)
- REG_DIG_H3
 - BME280, [93](#)
- REG_DIG_H4
 - BME280, [93](#)
- REG_DIG_H5
 - BME280, [93](#)
- REG_DIG_H6
 - BME280, [93](#)
- REG_DIG_P1_LSB
 - BME280, [93](#)
- REG_DIG_P1_MSB
 - BME280, [93](#)
- REG_DIG_P2_LSB
 - BME280, [93](#)
- REG_DIG_P2_MSB
 - BME280, [93](#)
- REG_DIG_P3_LSB
 - BME280, [93](#)
- REG_DIG_P3_MSB
 - BME280, [93](#)
- REG_DIG_P4_LSB
 - BME280, [93](#)
- REG_DIG_P4_MSB
 - BME280, [93](#)
- REG_DIG_P5_LSB
 - BME280, [93](#)
- REG_DIG_P5_MSB
 - BME280, [93](#)
- REG_DIG_P6_LSB
 - BME280, [93](#)
- REG_DIG_P6_MSB
 - BME280, [93](#)
- REG_DIG_P7_LSB
 - BME280, [93](#)
- REG_DIG_P7_MSB
 - BME280, [93](#)
- REG_DIG_P8_LSB
 - BME280, [93](#)
- REG_DIG_P8_MSB
 - BME280, [93](#)
- REG_DIG_P9_LSB
 - BME280, [93](#)
- REG_DIG_P9_MSB
 - BME280, [93](#)
- REG_DIG_T1_LSB
 - BME280, [93](#)
- REG_DIG_T1_MSB
 - BME280, [93](#)
- REG_DIG_T2_LSB
 - BME280, [93](#)
- REG_DIG_T2_MSB
 - BME280, [93](#)
- REG_DIG_T3_LSB
 - BME280, [93](#)
- REG_DIG_T3_MSB
 - BME280, [93](#)
- REG_HUMIDITY_MSB
 - BME280, [93](#)
- REG_PRESSURE_MSB
 - BME280, [93](#)
- REG_RESET
 - BME280, [93](#)
- REG_TEMPERATURE_MSB
 - BME280, [93](#)
- RES
 - Protocol, [35](#)
- reserved
 - INA3221::masken_reg_t, [129](#)
- RESET
 - Types, [72](#)
- reset
 - BME280, [94](#)

- Configuration Functions, 59
- INA3221::conf_reg_t, 108
- RMC_DATA_COMMAND
 - gps_commands.cpp, 180
- rmc_mutex_
 - NMEADData, 133
- rmc_tokens_
 - NMEADData, 133
- RTC clock, 40
 - bcd_to_bin, 50
 - bin_to_bcd, 49
 - clock_enable, 44
 - DS3231, 41
 - get_clock_sync_interval, 45
 - get_instance, 41
 - get_last_sync_time, 46
 - get_local_time, 46
 - get_time, 42
 - get_timezone_offset, 44
 - get_unix_time, 44
 - i2c_read_reg, 48
 - i2c_write_reg, 48
 - is_sync_needed, 47
 - read_temperature, 42
 - set_clock_sync_interval, 45
 - set_time, 41
 - set_timezone_offset, 45
 - set_unix_time, 43
 - sync_clock_with_gps, 47
 - update_last_sync_time, 46
- sample_interval_ms
 - TelemetryManager, 152
- satellites
 - TelemetryRecord, 156
- SATURDAY
 - DS3231.h, 166
- save_to_storage
 - Event Management, 55
- scan_connected_sensors
 - Sensors, 75
- SD_CARD_DETECT_PIN
 - pin_config.h, 226
- sd_card_init_status
 - SystemStateManager, 147
- sd_card_mounted
 - SystemStateManager, 147
- SD_CS_PIN
 - pin_config.h, 226
- SD_MISO_PIN
 - pin_config.h, 225
- SD_MOSI_PIN
 - pin_config.h, 226
- SD_SCK_PIN
 - pin_config.h, 226
- SD_SPI_PORT
 - pin_config.h, 225
- SECOND
 - Protocol, 36
- seconds
 - ds3231_data_t, 112
- send.cpp
 - send_frame_lora, 209
 - send_frame_uart, 209
 - send_message, 209
- send_frame_lora
 - communication.h, 200
 - send.cpp, 209
 - test_comand_handlers.cpp, 282
- send_frame_uart
 - communication.h, 199
 - send.cpp, 209
 - test_comand_handlers.cpp, 282
- send_message
 - communication.h, 199
 - send.cpp, 209
- Sensor Commands, 27
 - handle_get_sensor_data, 27
 - handle_get_sensor_list, 28
 - handle_sensor_config, 27
- sensor_
 - BH1750Wrapper, 90
 - BME280Wrapper, 106
- sensor_commands.cpp
 - SENSOR_CONFIGURE, 188
 - SENSOR_GROUP, 188
 - SENSOR_READ, 188
- SENSOR_CONFIGURE
 - sensor_commands.cpp, 188
- sensor_configure
 - Sensors, 74
- sensor_data_buffer
 - TelemetryManager, 152
- SENSOR_DATA_CSV_PATH
 - Telemetry Manager, 78
- SENSOR_GROUP
 - sensor_commands.cpp, 188
- sensor_init
 - Sensors, 74
- SENSOR_READ
 - sensor_commands.cpp, 188
- sensor_read_data
 - Sensors, 74
- SensorDataRecord, 138
 - humidity, 139
 - light, 139
 - pressure, 138
 - temperature, 138
 - timestamp, 138
- SensorDataTypeIdentifier
 - Sensors, 73
- Sensors, 72
 - ENVIRONMENT, 73
 - get_available_sensors, 75
 - get_sensor, 75
 - HUMIDITY, 74
 - LIGHT, 73

- LIGHT_LEVEL, [74](#)
- NONE, [73](#), [74](#)
- PRESSURE, [74](#)
- scan_connected_sensors, [75](#)
- sensor_configure, [74](#)
- sensor_init, [74](#)
- sensor_read_data, [74](#)
- SensorDataTypeIdentifier, [73](#)
- SensorType, [73](#)
- TEMPERATURE, [74](#)
- sensors
 - SensorWrapper, [141](#)
- SENSORS_I2C_PORT
 - pin_config.h, [225](#)
- SENSORS_I2C_SCL_PIN
 - pin_config.h, [225](#)
- SENSORS_I2C_SDA_PIN
 - pin_config.h, [225](#)
- SENSORS_POWER_ENABLE_PIN
 - pin_config.h, [225](#)
- SensorType
 - Sensors, [73](#)
- SensorWrapper, [139](#)
 - get_instance, [140](#)
 - sensors, [141](#)
 - SensorWrapper, [140](#)
- SEQ
 - Protocol, [35](#)
- SET
 - Protocol, [35](#)
- set_averaging_mode
 - Configuration Functions, [61](#)
- set_bootloader_reset_pending
 - SystemStateManager, [143](#)
- set_bus_conversion_time
 - Configuration Functions, [61](#)
- set_bus_measurement_disable
 - Configuration Functions, [61](#)
- set_bus_measurement_enable
 - Configuration Functions, [61](#)
- set_clock_sync_interval
 - RTC clock, [45](#)
- set_env_sensor_init_ok
 - SystemStateManager, [146](#)
- set_gps_collection_paused
 - SystemStateManager, [144](#)
- set_light_sensor_init_ok
 - SystemStateManager, [146](#)
- set_mode_continuous
 - Configuration Functions, [60](#)
- set_mode_power_down
 - Configuration Functions, [60](#)
- set_mode_triggered
 - Configuration Functions, [60](#)
- set_radio_init_ok
 - SystemStateManager, [145](#)
- set_sd_card_mounted
 - SystemStateManager, [144](#)
- set_shunt_conversion_time
 - Configuration Functions, [61](#)
- set_shunt_measurement_disable
 - Configuration Functions, [60](#)
- set_shunt_measurement_enable
 - Configuration Functions, [60](#)
- set_time
 - RTC clock, [41](#)
- set_timezone_offset
 - RTC clock, [45](#)
- set_uart_verbosity
 - SystemStateManager, [145](#)
- set_unix_time
 - RTC clock, [43](#)
- setUp
 - test_comand_handlers.cpp, [282](#)
 - test_frame_send.cpp, [291](#)
- shunt_conv_time
 - INA3221::conf_reg_t, [107](#)
- shunt_sum_alert
 - INA3221::masken_reg_t, [128](#)
- shunt_sum_en_ch1
 - INA3221::masken_reg_t, [129](#)
- shunt_sum_en_ch2
 - INA3221::masken_reg_t, [129](#)
- shunt_sum_en_ch3
 - INA3221::masken_reg_t, [129](#)
- SHUNT_VOLTAGE_LSB_UV
 - INA3221.h, [237](#)
- SHUTDOWN
 - Event Management, [53](#)
- SILENT
 - utils.h, [273](#)
- SOLAR_ACTIVE
 - Event Management, [53](#)
- SOLAR_CURRENT_THRESHOLD
 - PowerManager, [136](#)
- SOLAR_INACTIVE
 - Event Management, [53](#)
- speed
 - TelemetryRecord, [156](#)
- spi_output_buffer
 - hardware_mock.cpp, [293](#)
 - hardware_mock.h, [296](#)
- SPI_PORT
 - pin_config.h, [227](#)
- split_and_send_message
 - communication.h, [200](#)
- splitString
 - Location, [57](#)
- Storage, [76](#)
 - fs_init, [76](#)
 - fs_stop, [76](#)
- Storage Commands, [29](#)
 - handle_list_files, [29](#)
 - handle_mount, [29](#)
- storage_commands.cpp
 - LIST_FILES_COMMAND, [192](#)

- MOUNT_COMMAND, 192
 - STORAGE_GROUP, 192
- STORAGE_GROUP
 - storage_commands.cpp, 192
- string_to_operation_type
 - Utility Converters, 39
- SUNDAY
 - DS3231.h, 166
- SX1278_CS
 - pin_config.h, 226
- SX1278_MISO
 - pin_config.h, 226
- SX1278_MOSI
 - pin_config.h, 226
- SX1278_SCK
 - pin_config.h, 226
- sync_clock_with_gps
 - RTC clock, 47
- sync_interval_minutes_
 - DS3231, 111
- SYSTEM
 - Event Management, 53
- System State Manager, 77
- system_voltage
 - TelemetryRecord, 154
- SystemEvent
 - Event Management, 53
- SystemStateManager, 141
 - env_sensor_init_status, 148
 - get_instance, 143
 - get_uart_verbosity, 145
 - gps_collection_paused, 147
 - is_bootloader_reset_pending, 143
 - is_env_sensor_init_ok, 146
 - is_gps_collection_paused, 144
 - is_light_sensor_init_ok, 146
 - is_radio_init_ok, 145
 - is_sd_card_mounted, 144
 - light_sensor_init_status, 148
 - mutex_, 148
 - operator=, 143
 - pending_bootloader_reset, 147
 - radio_init_status, 147
 - sd_card_init_status, 147
 - sd_card_mounted, 147
 - set_bootloader_reset_pending, 143
 - set_env_sensor_init_ok, 146
 - set_gps_collection_paused, 144
 - set_light_sensor_init_ok, 146
 - set_radio_init_ok, 145
 - set_sd_card_mounted, 144
 - set_uart_verbosity, 145
 - SystemStateManager, 143
 - uart_verbosity, 147
- t_fine
 - BME280, 98
- tearDown
 - test_comand_handlers.cpp, 283
- test_frame_send.cpp, 291
- Telemetry Buffer Commands, 30
 - handle_get_last_sensor_record, 31
 - handle_get_last_telemetry_record, 30
- Telemetry Manager, 77
 - collect_gps_telemetry, 80
 - collect_power_telemetry, 79
 - collect_sensor_telemetry, 80
 - collect_telemetry, 81
 - DEFAULT_FLUSH_THRESHOLD, 79
 - DEFAULT_SAMPLE_INTERVAL_MS, 78
 - emit_power_events, 80
 - flush_telemetry, 81
 - get_last_sensor_record_csv, 83
 - get_last_telemetry_record_csv, 83
 - init, 81
 - is_telemetry_collection_time, 82
 - is_telemetry_flush_time, 82
 - SENSOR_DATA_CSV_PATH, 78
 - TELEMETRY_CSV_PATH, 78
 - TelemetryManager, 80
 - to_csv, 79
- telemetry_buffer
 - TelemetryManager, 152
- telemetry_buffer_count
 - TelemetryManager, 152
- TELEMETRY_BUFFER_SIZE
 - TelemetryManager, 151
- telemetry_buffer_write_index
 - TelemetryManager, 152
- telemetry_commands.cpp
 - GET_LAST_SENSOR_RECORD_COMMAND, 195
 - GET_LAST_TELEMETRY_RECORD_COMMAND, 194
 - TELEMETRY_GROUP, 194
- TELEMETRY_CSV_PATH
 - Telemetry Manager, 78
- TELEMETRY_GROUP
 - telemetry_commands.cpp, 194
- telemetry_mutex
 - TelemetryManager, 153
- TelemetryManager, 148
 - ~TelemetryManager, 150
 - DEFAULT_FLUSH_THRESHOLD, 152
 - DEFAULT_SAMPLE_INTERVAL_MS, 151
 - flush_sensor_data, 150
 - flush_threshold, 152
 - get_instance, 150
 - get_last_sensor_record, 151
 - get_last_telemetry_record, 151
 - get_telemetry_buffer_count, 151
 - get_telemetry_buffer_write_index, 151
 - sample_interval_ms, 152
 - sensor_data_buffer, 152
 - Telemetry Manager, 80
 - telemetry_buffer, 152
 - telemetry_buffer_count, 152

- TELEMETRY_BUFFER_SIZE, 151
- telemetry_buffer_write_index, 152
- telemetry_mutex, 153
- TelemetryRecord, 153
 - altitude, 156
 - battery_voltage, 154
 - build_version, 154
 - charge_current_solar, 154
 - charge_current_usb, 154
 - course, 156
 - date, 156
 - discharge_current, 155
 - fix_quality, 156
 - lat_dir, 155
 - latitude, 155
 - lon_dir, 155
 - longitude, 155
 - satellites, 156
 - speed, 156
 - system_voltage, 154
 - time, 155
 - timestamp, 154
- TEMPERATURE
 - Sensors, 74
- temperature
 - SensorDataRecord, 138
- TEMPERATURE_OVERSAMPLING
 - BME280, 94
- test/comms/commands/test_clock_commands.cpp, 279
- test/comms/commands/test_diagnostic_commands.cpp, 279, 280
- test/comms/commands/test_event_commands.cpp, 281
- test/comms/commands/test_gps_commands.cpp, 281
- test/comms/commands/test_power_commands.cpp, 281
- test/comms/commands/test_sensor_commands.cpp, 281
- test/comms/commands/test_storage_commands.cpp, 281
- test/comms/commands/test_telemetry_commands.cpp, 282
- test/comms/test_comand_handlers.cpp, 282, 284
- test/comms/test_converters.cpp, 284, 286
- test/comms/test_frame_build.cpp, 286, 288
- test/comms/test_frame_coding.cpp, 288, 289
- test/comms/test_frame_common.h, 290
- test/comms/test_frame_send.cpp, 291, 292
- test/mocks/hardware_mocks.cpp, 292, 294
- test/mocks/hardware_mocks.h, 294, 296
- test/test_runner.cpp, 296, 300
- test_comand_handlers.cpp
 - last_frame_sent, 283
 - lora_send_called, 283
 - send_frame_lora, 282
 - send_frame_uart, 282
 - setUp, 282
 - tearDown, 283
 - test_command_handler_get_operation, 283
 - test_command_handler_invalid_operation, 283
 - test_command_handler_set_operation, 283
 - uart_send_called, 283
- test_command_handler_get_operation
 - test_comand_handlers.cpp, 283
 - test_runner.cpp, 299
- test_command_handler_invalid_operation
 - test_comand_handlers.cpp, 283
 - test_runner.cpp, 299
- test_command_handler_set_operation
 - test_comand_handlers.cpp, 283
 - test_runner.cpp, 299
- test_converters.cpp
 - test_exception_type_conversion, 285
 - test_hex_string_conversion, 285
 - test_operation_type_conversion, 285
 - test_value_unit_type_conversion, 285
- test_diagnostic_commands.cpp
 - test_handle_enter_bootloader_mode, 280
 - test_handle_get_build_version, 279
 - test_handle_get_commands_list, 279
 - test_handle_verbosity, 279
- test_error_code_conversion
 - test_runner.cpp, 300
- test_exception_type_conversion
 - test_converters.cpp, 285
 - test_runner.cpp, 299
- test_frame_build.cpp
 - test_frame_build_err, 286
 - test_frame_build_get, 287
 - test_frame_build_res, 287
 - test_frame_build_seq, 287
 - test_frame_build_set, 287
 - test_frame_build_val, 286
- test_frame_build_err
 - test_frame_build.cpp, 286
 - test_runner.cpp, 298
- test_frame_build_get
 - test_frame_build.cpp, 287
 - test_runner.cpp, 297
- test_frame_build_res
 - test_frame_build.cpp, 287
 - test_runner.cpp, 298
- test_frame_build_seq
 - test_frame_build.cpp, 287
 - test_runner.cpp, 298
- test_frame_build_set
 - test_frame_build.cpp, 287
 - test_runner.cpp, 298
- test_frame_build_val
 - test_frame_build.cpp, 286
 - test_runner.cpp, 298
- test_frame_coding.cpp
 - test_frame_decode_basic, 289
 - test_frame_decode_invalid_header, 289
 - test_frame_encode_basic, 289
- test_frame_common.h
 - create_test_frame, 290

- test_frame_decode_basic
 - test_frame_coding.cpp, [289](#)
 - test_runner.cpp, [297](#)
- test_frame_decode_invalid_header
 - test_frame_coding.cpp, [289](#)
 - test_runner.cpp, [297](#)
- test_frame_encode_basic
 - test_frame_coding.cpp, [289](#)
 - test_runner.cpp, [297](#)
- test_frame_send.cpp
 - setUp, [291](#)
 - tearDown, [291](#)
 - test_send_frame_uart, [291](#)
- test_handle_enter_bootloader_mode
 - test_diagnostic_commands.cpp, [280](#)
 - test_runner.cpp, [300](#)
- test_handle_get_build_version
 - test_diagnostic_commands.cpp, [279](#)
 - test_runner.cpp, [299](#)
- test_handle_get_commands_list
 - test_diagnostic_commands.cpp, [279](#)
 - test_runner.cpp, [299](#)
- test_handle_verbosity
 - test_diagnostic_commands.cpp, [279](#)
 - test_runner.cpp, [300](#)
- test_hex_string_conversion
 - test_converters.cpp, [285](#)
 - test_runner.cpp, [299](#)
- test_operation_type_conversion
 - test_converters.cpp, [285](#)
 - test_runner.cpp, [298](#)
- test_runner.cpp
 - main, [300](#)
 - test_command_handler_get_operation, [299](#)
 - test_command_handler_invalid_operation, [299](#)
 - test_command_handler_set_operation, [299](#)
 - test_error_code_conversion, [300](#)
 - test_exception_type_conversion, [299](#)
 - test_frame_build_err, [298](#)
 - test_frame_build_get, [297](#)
 - test_frame_build_res, [298](#)
 - test_frame_build_seq, [298](#)
 - test_frame_build_set, [298](#)
 - test_frame_build_val, [298](#)
 - test_frame_decode_basic, [297](#)
 - test_frame_decode_invalid_header, [297](#)
 - test_frame_encode_basic, [297](#)
 - test_handle_enter_bootloader_mode, [300](#)
 - test_handle_get_build_version, [299](#)
 - test_handle_get_commands_list, [299](#)
 - test_handle_verbosity, [300](#)
 - test_hex_string_conversion, [299](#)
 - test_operation_type_conversion, [298](#)
 - test_value_unit_type_conversion, [298](#)
- test_send_frame_uart
 - test_frame_send.cpp, [291](#)
- test_value_unit_type_conversion
 - test_converters.cpp, [285](#)
- test_runner.cpp, [298](#)
- TEXT
 - Protocol, [36](#)
- THURSDAY
 - DS3231.h, [166](#)
- TIME
 - clock_commands.cpp, [169](#)
- time
 - TelemetryRecord, [155](#)
- timestamp
 - event_manager.h, [216](#)
 - EventLog, [115](#)
 - SensorDataRecord, [138](#)
 - TelemetryRecord, [154](#)
- TIMEZONE_OFFSET
 - clock_commands.cpp, [169](#)
- timezone_offset_minutes_
 - DS3231, [111](#)
- timing_ctrl_alert
 - INA3221::masken_reg_t, [127](#)
- to_csv
 - Telemetry Manager, [79](#)
- TUESDAY
 - DS3231.h, [166](#)
- Types, [71](#)
 - CONTINUOUS_HIGH_RES_MODE, [72](#)
 - CONTINUOUS_HIGH_RES_MODE_2, [72](#)
 - CONTINUOUS_LOW_RES_MODE, [72](#)
 - Mode, [72](#)
 - ONE_TIME_HIGH_RES_MODE, [72](#)
 - ONE_TIME_HIGH_RES_MODE_2, [72](#)
 - ONE_TIME_LOW_RES_MODE, [72](#)
 - POWER_ON, [72](#)
 - RESET, [72](#)
 - UNCONFIGURED_POWER_DOWN, [72](#)
- UART
 - Protocol, [36](#)
- UART_ERROR
 - Event Management, [54](#)
- uart_mutex
 - utils.cpp, [271](#)
- uart_output_buffer
 - hardwaremocks.cpp, [293](#)
 - hardwaremocks.h, [295](#)
- uart_print
 - utils.cpp, [270](#)
 - utils.h, [274](#)
- uart_send_called
 - test_comand_handlers.cpp, [283](#)
- uart_verbosity
 - SystemStateManager, [147](#)
- UNCONFIGURED_POWER_DOWN
 - Types, [72](#)
- UNDEFINED
 - Protocol, [36](#)
- unit
 - Frame, [120](#)
- UNKNOWN_ERROR

- Protocol, [35](#)
- update_gga_tokens
 - NMEADData, [132](#)
- update_last_sync_time
 - RTC clock, [46](#)
- update_rmc_tokens
 - NMEADData, [131](#)
- USB_CONNECTED
 - Event Management, [53](#)
- USB_CURRENT_THRESHOLD
 - PowerManager, [136](#)
- USB_DISCONNECTED
 - Event Management, [53](#)
- Utility Converters, [38](#)
 - error_code_to_string, [39](#)
 - exception_type_to_string, [38](#)
 - hex_string_to_bytes, [39](#)
 - operation_type_to_string, [38](#)
 - string_to_operation_type, [39](#)
 - value_unit_type_to_string, [38](#)
- utils.cpp
 - get_level_color, [270](#)
 - get_level_prefix, [270](#)
 - uart_mutex, [271](#)
 - uart_print, [270](#)
- utils.h
 - ANSI_BLUE, [273](#)
 - ANSI_GREEN, [273](#)
 - ANSI_RED, [273](#)
 - ANSI_RESET, [273](#)
 - ANSI_YELLOW, [273](#)
 - DEBUG, [273](#)
 - ERROR, [273](#)
 - INFO, [273](#)
 - SILENT, [273](#)
 - uart_print, [274](#)
 - VerbosityLevel, [273](#)
 - WARNING, [273](#)
- VAL
 - Protocol, [35](#)
- value
 - Frame, [120](#)
- value_unit_type_to_string
 - Utility Converters, [38](#)
- ValueUnit
 - Protocol, [35](#)
- VerbosityLevel
 - utils.h, [273](#)
- VOLT
 - Protocol, [36](#)
- VOLTAGE_BATTERY
 - power_commands.cpp, [184](#)
- VOLTAGE_MAIN
 - power_commands.cpp, [184](#)
- warn_alert_ch1
 - INA3221::masken_reg_t, [128](#)
- warn_alert_ch2
 - INA3221::masken_reg_t, [128](#)
- warn_alert_ch3
 - INA3221::masken_reg_t, [127](#)
- warn_alert_latch_en
 - INA3221::masken_reg_t, [128](#)
- WARNING
 - utils.h, [273](#)
- WATCHDOG_RESET
 - Event Management, [53](#)
- WEDNESDAY
 - DS3231.h, [166](#)
- write8
 - BH1750 Light Sensor, [70](#)
- WRITE_ONLY
 - Protocol, [35](#)
- write_register
 - BME280, [96](#)
- writeIndex
 - EventManager, [118](#)
- year
 - ds3231_data_t, [113](#)