Sensor Board

All about the sensor board

Overview

A 480MHz Swiss Army knife of the embedded team☮

Table of Contents

Purpose

The Sensor Board is a specialized embedded system designed to acquire environmental and motion data from multiple sensors. It integrates position (GNSS), water quality (pH), motion (IMU), force (load cells), and pressure measurements, transmitting all data over Ethernet using Protocol Buffer encoding.

Hardware Platform

- **Microcontroller:** STM32H753ZI (Nucleo board)
- **Real-Time OS:** FreeRTOS with CMSIS-RTOS V2
- **Communication:**
  - Ethernet (LAN8742 PHY)
  - UART (Multiple sensor connections)
  - I2C/SPI (IMU, pressure sensors)
- **Memory:** 64KB heap allocated for FreeRTOS
- **Clock:** 480 MHz ARM Cortex-M7

Key Board Features

- Multi-sensor fusion with independent sensor threads
- Network integration via UDP/Ethernet with Protobuf
- Real-time logging to UART (115200 baud)
- MAC address filtering for selective communication
- Packet dispatcher for incoming control signals
- LED status indicators (Green, Blue, Red)
- Heap monitoring with critical threshold alerts

Core Sensors Integrated

1. **GPS (GNSS)** - Global positioning and velocity
2. **IMU** - 3-axis acceleration, rotation, magnetic field
3. **pH Sensor** - Water quality measurement
4. **Load Cells** - Force measurement (×2)
5. **Pressure Sensors** - Pressure measurement (×2)

STM32CubeMX Sensor Configuration

This page documents the current STM32CubeMX configuration for the Sensor Board firmware and explains how to extend it for sensor interfaces (UART/I2C/SPI/ADC) in a way that is safe for code generation.

- **IOC File**: `components/sensor_board/firmware/firmware.ioc`
- **Generated HAL Init Files**: `components/sensor_board/firmware/Core/Src/`
- **Application Entry**: `src/sensor_board/main.c`

Current CubeMX Snapshot

Item	Value
MCU	STM32H753ZIT6 (NUCLEO-H753ZI)
CubeMX Version	6.15.0
STM32Cube FW Package	STM32Cube FW_H7 v1.12.1
Toolchain	Makefile + GCC
Post-generation Script	`../../../scripts/post_code_generation.bash`

Enabled CubeMX Components

- **CORTEX_M7** (I-Cache/D-Cache enabled, MPU configured)
- **ETH** (RMII mode)
- **LWIP** (Static IP, DHCP disabled)
- **FREERTOS** (CMSIS-RTOS v2, default task generated)
- **TIM1** (base timer)
- **SYS/NVIC/RCC** base platform configuration

Clock and Core Setup

Clock Configuration (from IOC)

Parameter	Value
Clock Source	HSE 8 MHz -> PLL
SYSCLK	72 MHz
APB1	36 MHz (DIV2)
APB2/APB3/APB4	72 MHz
TIM1 Clock	72 MHz

Cortex-M7 / MPU

- Instruction cache: enabled
- Data cache: enabled
- MPU region at `0x30000000`, size `32KB`, shareable, non-cacheable

Pinout and Peripheral Mapping

Ethernet (RMII)

Signal	Pin
ETH_REF_CLK	PA1
ETH_MDIO	PA2
ETH_CRS_DV	PA7
ETH_MDC	PC1
ETH_RXD0	PC4
ETH_RXD1	PC5
ETH_TX_EN	PG11
ETH_TXD0	PG13
ETH_TXD1	PB13

Serial / COM

Signal	Pin	Note
USART1_TX	PA9	Configured in generated `MX_GPIO_Init()`

USART1_RX	PA10	Configured in generated `MX_GPIO_Init()`
USART3_TX	PD8	Used by NUCLEO COM path (`MX_USART3_Init` in app init)
USART3_RX	PD9	Used by NUCLEO COM path (`MX_USART3_Init` in app init)

Board IO / Misc

Pin	Mode	Typical Use
PC13	GPIO Input	User button
PB0	GPIO Output	Board output line
PB7	GPIO Output	Board output line
PB14	GPIO Output	Board output line
PH0 / PH1	HSE oscillator	System clock source
PC14 / PC15	LSE oscillator	Low-speed oscillator

Timer, RTOS, and Interrupts

TIM1 Base Timer

```
htim1.Init.Prescaler = 6400 - 1;
htim1.Init.Period = 10000;
```

With a 72 MHz timer clock, this gives an update period near 0.89 s.

Interrupt Priorities (Key Entries)

IRQ	Priority	Notes
ETH_IRQn	15	Ethernet/LwIP path
TIM1_UP_IRQn	12	TIM1 update interrupt
TIM2_IRQn	7	HAL tick time base
EXTI15_10_IRQn	6	External interrupt group

Sensor Interface Status

What Is Already Configured in CubeMX

- Networking stack and RMII pinout
- Base timer and RTOS scaffolding
- Basic UART-capable pins and NUCLEO COM integration path

What Is Not Yet Fully Modeled in CubeMX (TO-DO ONCE sensors retrieved and assembled)

- Dedicated ADC channels for analog sensors (pH, load cell, pressure)
- Dedicated I2C/SPI buses for IMU and pressure variants
- Explicit sensor-specific pin labels and alternate-function assignments

Important: Current sensor drivers include placeholders for hardware access in multiple modules. When bringing up physical sensors, add the corresponding CubeMX peripherals first, then update the sensor drivers to use generated handles.

Recommended Workflow

1. Open `components/sensor_board/firmware/firmware.ioc` in STM32CubeMX.
2. Add required peripherals for the target sensor like this:

```
GPS        -> USARTx (baud/parity/stop bits to match module)
IMU        -> I2Cx or SPIx (+ optional DRDY INT GPIO)
pH         -> ADCx channel (sampling time, resolution)
Load Cell  -> ADCx channel(s) or external ADC interface
Pressure   -> ADCx or I2Cx/SPIx (depends on sensor part)
```

1. Assign and lock pins in Pinout view; avoid overlap with RMII and COM pins.
2. Configure clocks for new peripherals in Clock Configuration.
3. Set NVIC priorities for new ISR sources so Ethernet/RTOS timing remains stable.
4. Generate code with **Keep User Code** enabled.
5. Rebuild using PlatformIO and validate startup + sensor polling.

Conflict To Look Out for Before Saving .ioc file

- No conflict with ETH RMII pins (PA1, PA2, PA7, PC1, PC4, PC5, PG11, PG13, PB13)
- No conflict with debug/COM path (PA9/PA10 and PD8/PD9)

- No conflict with oscillator pins (PH0, PH1, PC14, PC15)

Related Pages

- Configuration
- Reference
- Sensor Basics Utility Library

Sensor Basics Utility Library

These are basic features that could be used if required... But was something made in "spare time"

Source Code Location

Files:

- `components/sensor_board/sensor_basics/sensor_basics.h` - Function declarations and documentation
- `components/sensor_board/sensor_basics/sensor_basics.c` - Implementation

Dependencies:

- `result.h` - Standard result/error code definitions
- `stdint.h` - Integer type definitions

---

pH Sensor Functions

validate_ph_value()

Validates if a pH value is within the acceptable range (0-14).

```
/**
 * @brief Validates if a pH value is within the acceptable range (0-14).
 * @param ph_value The pH value to validate.
 * @return RESULT_OK if the value is valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_ph_value(float ph_value);
```

Implementation:

```
result_t validate_ph_value(float ph_value) {
    if (ph_value >= 0.0f && ph_value <= 14.0f) {
        return RESULT_OK;
```

```
    }
    return RESULT_ERR_INVALID_DATA;
}
```

Parameters:

- `ph_value` - Float value to validate (typical range: 0.0 to 14.0)

Return Values:

- `RESULT_OK` - Value is within valid range
- `RESULT_ERR_INVALID_DATA` - Value is outside 0-14 range

Usage Example:

```
float ph_reading = ph_sensor.ph_value;

if (validate_ph_value(ph_reading) == RESULT_OK) {
    LOG_INFO("pH sensor valid: %.2f", ph_reading);
    diagnostics.ph_sensor.state = SENSOR_OPERATING;
} else {
    LOG_ERROR("pH out of range: %.2f", ph_reading);
    diagnostics.ph_sensor.state = SENSOR_ERROR;
    diagnostics.ph_sensor.error_code = PHErrorCode_PH_INVALID_DATA;
}
```

Temperature Conversion Functions

celsius_to_fahrenheit()

Converts temperature from Celsius to Fahrenheit.

```
/**
 * @brief Converts temperature from Celsius to Fahrenheit.
 * @param celsius The temperature in Celsius.
 * @param fahrenheit Pointer to a float where the converted Fahrenheit temperature will be
stored.
 * @return RESULT_OK on success, or RESULT_ERR_INVALID_ARG if fahrenheit is NULL.
 */
```

```
result_t celsius_to_fahrenheit(float celsius, float *fahrenheit);
```

Conversion Formula: °F = (°C × 9/5) + 32

Parameters:

- `celsius` - Input temperature in Celsius
- `fahrenheit` - Pointer to output variable (must not be NULL)

Return Values:

- `RESULT_OK` - Conversion successful
- `RESULT_ERR_INVALID_ARG` - fahrenheit pointer is NULL

Status: Currently commented out in implementation

Usage Example:

```
float temp_celsius = 25.0f;
float temp_fahrenheit;

if (celsius_to_fahrenheit(temp_celsius, &temp_fahrenheit) == RESULT_OK) {
    LOG_INFO("Temperature: %.1f°C = %.1f°F", temp_celsius, temp_fahrenheit);
}
```

fahrenheit_to_celsius()

Converts temperature from Fahrenheit to Celsius.

```
/**
 * @brief Converts temperature from Fahrenheit to Celsius.
 * @param fahrenheit The temperature in Fahrenheit.
 * @param celsius Pointer to a float where the converted Celsius temperature will be stored.
 * @return RESULT_OK on success, or RESULT_ERR_INVALID_ARG if celsius is NULL.
 */
result_t fahrenheit_to_celsius(float fahrenheit, float *celsius);
```

Conversion Formula: °C = (°F - 32) × 5/9

Status: Currently commented out in implementation

IMU (Accelerometer) Functions

validate_accelerometer_value()

Validates if a single accelerometer value is within the typical range.

```
/**
 * @brief Validates if an accelerometer value is within the typical range.
 * @param accel_value The accelerometer value to validate.
 * @return RESULT_OK if the value is valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_accelerometer_value(float accel_value);
```

Implementation:

```
result_t validate_accelerometer_value(float accel_value) {
    if (accel_value >= -160.0f && accel_value <= 160.0f) {
        return RESULT_OK;
    }
    return RESULT_ERR_INVALID_DATA;
}
```

Parameters:

- `accel_value` - Single axis acceleration value in m/s²

Valid Range: -160.0 to +160.0 m/s² (typical ±16g sensor range)

Return Values:

- `RESULT_OK` - Value is within valid range
- `RESULT_ERR_INVALID_DATA` - Value exceeds acceptable limits

Usage Example:

```
float accel_x = imu_data.accel[0];

if (validate_accelerometer_value(accel_x) == RESULT_OK) {
    LOG_DEBUG("Accel X valid: %.2f m/s²", accel_x);
} else {
    LOG_ERROR("Accel X out of range: %.2f m/s²", accel_x);
```

```
}
```

validate_imu_data()

Validates all three axes of accelerometer data simultaneously.

```
/**
 * @brief Validates all three axes of accelerometer data.
 * @param accel_x The acceleration value for the X-axis.
 * @param accel_y The acceleration value for the Y-axis.
 * @param accel_z The acceleration value for the Z-axis.
 * @return RESULT_OK if all values are valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_imu_data(float accel_x, float accel_y, float accel_z);
```

Implementation:

```
result_t validate_imu_data(float accel_x, float accel_y, float accel_z) {
    TRY(validate_accelerometer_value(accel_x));
    TRY(validate_accelerometer_value(accel_y));
    TRY(validate_accelerometer_value(accel_z));
    return RESULT_OK;
}
```

Parameters:

- `accel_x` - X-axis acceleration (m/s²)
- `accel_y` - Y-axis acceleration (m/s²)
- `accel_z` - Z-axis acceleration (m/s²)

Return Values:

- `RESULT_OK` - All three axes within valid range
- `RESULT_ERR_INVALID_DATA` - Any axis exceeds acceptable limits

Usage Example:

```
if (validate_imu_data(imu_data.accel[0], imu_data.accel[1], imu_data.accel[2]) == RESULT_OK) {
    LOG_INFO("IMU acceleration valid");
    diagnostics.imu_sensor.state = SENSOR_OPERATING;
} else {
    LOG_ERROR("IMU acceleration out of range");
```

```
    diagnostics.imu_sensor.state = SENSOR_ERROR;
}
```

Pressure Conversion Functions

bar_to_psi()

Converts pressure from bar to psi (pounds per square inch).

```
/**
 * @brief Converts pressure from bar to psi.
 * @param bar The pressure in bar.
 * @param psi Pointer to a float where the converted psi pressure will be stored.
 * @return RESULT_OK on success, or RESULT_ERR_INVALID_ARG if psi is NULL.
 */
result_t bar_to_psi(float bar, float *psi);
```

Conversion Formula: psi = bar × 14.5038

Parameters:

- `bar` - Pressure in bar
- `psi` - Pointer to output variable (must not be NULL)

Return Values:

- `RESULT_OK` - Conversion successful
- `RESULT_ERR_INVALID_ARG` - psi pointer is NULL

Status: Currently commented out in implementation

Usage Example:

```
float pressure_bar = 5.0f;
float pressure_psi;

if (bar_to_psi(pressure_bar, &pressure_psi) == RESULT_OK) {
    LOG_INFO("Pressure: %.2f bar = %.2f psi", pressure_bar, pressure_psi);
}
```

psi_to_bar()

Converts pressure from psi to bar.

```
/**
 * @brief Converts pressure from psi to bar.
 * @param psi The pressure in psi.
 * @param bar Pointer to a float where the converted bar pressure will be stored.
 * @return RESULT_OK on success, or RESULT_ERR_INVALID_ARG if bar is NULL.
 */
result_t psi_to_bar(float psi, float *bar);
```

Conversion Formula: bar = psi ÷ 14.5038

Status: Currently commented out in implementation

GPS Functions

validate_gps_latitude()

Validates GPS latitude value is within valid range.

```
/**
 * @brief Validates GPS latitude value.
 * @param latitude The latitude value to validate (-90 to +90 degrees).
 * @return RESULT_OK if the value is valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_gps_latitude(double latitude);
```

Implementation:

```
result_t validate_gps_latitude(double latitude) {
    if (latitude >= -90.0 && latitude <= 90.0) {
        return RESULT_OK;
    }
    return RESULT_ERR_INVALID_DATA;
}
```

Parameters:

- `latitude` - Latitude in degrees (double precision)

Valid Range: -90.0 to +90.0 degrees

- Negative = South
- Positive = North
- 0° = Equator

Return Values:

- `RESULT_OK` - Value is within valid range
- `RESULT_ERR_INVALID_DATA` - Value outside ±90 degrees

validate_gps_longitude()

Validates GPS longitude value is within valid range.

```
/**
 * @brief Validates GPS longitude value.
 * @param longitude The longitude value to validate (-180 to +180 degrees).
 * @return RESULT_OK if the value is valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_gps_longitude(double longitude);
```

Implementation:

```
result_t validate_gps_longitude(double longitude) {
    if (longitude >= -180.0 && longitude <= 180.0) {
        return RESULT_OK;
    }
    return RESULT_ERR_INVALID_DATA;
}
```

Parameters:

- `longitude` - Longitude in degrees (double precision)

Valid Range: -180.0 to +180.0 degrees

- Negative = West
- Positive = East
- 0° = Prime Meridian
- ±180° = International Date Line

Return Values:

- `RESULT_OK` - Value is within valid range
- `RESULT_ERR_INVALID_DATA` - Value outside ±180 degrees

validate_gps_hdop()

Validates GPS Horizontal Dilution of Precision (HDOP) value.

```
/**
 * @brief Validates GPS HDOP value.
 * @param hdop The HDOP value to validate (0-50 typical range).
 * @return RESULT_OK if the value is valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_gps_hdop(float hdop);
```

Implementation:

```
result_t validate_gps_hdop(float hdop) {
    if (hdop >= 0.0f && hdop <= 50.0f) {
        return RESULT_OK;
    }
    return RESULT_ERR_INVALID_DATA;
}
```

Parameters:

- `hdop` - Horizontal dilution of precision value

Valid Range: 0.0 to 50.0

HDOP Quality Interpretation:

- <1 - Ideal
- 1-2 - Excellent
- 2-5 - Good
- 5-10 - Moderate
- 10-20 - Fair
- >20 - Poor

Return Values:

- `RESULT_OK` - Value is within valid range

- `RESULT_ERR_INVALID_DATA` - Value exceeds acceptable limits

validate_gps_satellite_count()

Validates GPS satellite count is within valid range.

```
/**
 * @brief Validates GPS satellite count.
 * @param satellites The number of satellites to validate.
 * @return RESULT_OK if the value is valid, RESULT_ERR_INVALID_DATA otherwise.
 */
result_t validate_gps_satellite_count(int32_t satellites);
```

Implementation:

```
result_t validate_gps_satellite_count(int32_t satellites) {
    if (satellites >= 0 && satellites <= 30) {
        return RESULT_OK;
    }
    return RESULT_ERR_INVALID_DATA;
}
```

Parameters:

- `satellites` - Number of satellites in view

Valid Range: 0 to 30 satellites

Satellite Count Guidance:

- 0-3 - No/poor fix possible
- 4-5 - 3D fix possible
- 6-9 - Good coverage
- 10+ - Excellent coverage

Return Values:

- `RESULT_OK` - Value is within valid range
- `RESULT_ERR_INVALID_DATA` - Negative value or exceeds 30

Usage Example (Full GPS Validation):

```
if (validate_gps_latitude(gps_data.latitude) == RESULT_OK &&
    validate_gps_longitude(gps_data.longitude) == RESULT_OK &&
    validate_gps_hdop(gps_data.hdop) == RESULT_OK &&
    validate_gps_satellite_count(gps_data.satellites) == RESULT_OK) {

    LOG_INFO("GPS fix valid: %.6f, %.6f (sats=%d)",
             gps_data.latitude, gps_data.longitude, gps_data.satellites);
    diagnostics.gps_sensor_1.state = SENSOR_OPERATING;
} else {
    LOG_WARNING("GPS validation failed");
    diagnostics.gps_sensor_1.state = SENSOR_ERROR;
}
```

Error Handling Pattern

All validation functions follow a consistent pattern:

```
// Check sensor data
if (validate_<sensor>_<field>(value) == RESULT_OK) {
    // Data is valid - use it
    diagnostics.<sensor>.state = SENSOR_OPERATING;
} else {
    // Data is invalid - set error state
    diagnostics.<sensor>.state = SENSOR_ERROR;
    diagnostics.<sensor>.error_code = <ERROR_CODE>_INVALID_DATA;
}
```

TRY Macro Usage:

The implementation uses a `TRY()` macro for error propagation (from result.h):

```
// In validate_imu_data()
result_t validate_imu_data(float accel_x, float accel_y, float accel_z) {
    TRY(validate_accelerometer_value(accel_x));  // Return on error
    TRY(validate_accelerometer_value(accel_y));  // Return on error
    TRY(validate_accelerometer_value(accel_z));  // Return on error
    return RESULT_OK;
}
```

Implementation Status

Currently Implemented (Active):

- ✓ validate_ph_value()
- ✓ validate_accelerometer_value()
- ✓ validate_imu_data()
- ✓ validate_gps_latitude()
- ✓ validate_gps_longitude()
- ✓ validate_gps_hdop()
- ✓ validate_gps_satellite_count()

Currently Commented Out (Inactive):

- ⊘ celsius_to_fahrenheit() - Declared but not implemented
- ⊘ fahrenheit_to_celsius() - Declared but not implemented
- ⊘ bar_to_psi() - Declared but not implemented
- ⊘ psi_to_bar() - Declared but not implemented

Note: Temperature and pressure conversions are stubbed out in the current implementation. They can be enabled by uncommenting the implementation in sensor_basics.c if needed for future features.

Testing

Test Suite Location: `test/sensor_board/test_sensor_basics/`

Building Tests:

```
// Run all sensor_basics tests
pio test -e sensor_board -f test_sensor_basics


// Run with verbose output
pio test -e sensor_board -f test_sensor_basics -v
```

Test Coverage:

- Boundary value testing (min/max ranges)
- Edge cases (0 values, extreme values)
- Valid range acceptance

- Invalid range rejection
- NULL pointer handling for conversion functions

Integration in Main Application

Typical Usage in main.c:

```
// After polling a sensor
result_t poll_result = poll_gps_sensor(&gps_data);

if (poll_result == RESULT_OK) {
    // Validate all GPS fields before using
    if (validate_gps_latitude(gps_data.latitude) == RESULT_OK &&
        validate_gps_longitude(gps_data.longitude) == RESULT_OK) {

        diagnostics.gps_sensor_1.state = SENSOR_OPERATING;
        // Safe to use: gps_data.latitude, gps_data.longitude

    } else {
        diagnostics.gps_sensor_1.state = SENSOR_ERROR;
        diagnostics.gps_sensor_1.error_code = GPS_INVALID_DATA;
    }
} else if (poll_result == RESULT_ERR_COMMS) {
    diagnostics.gps_sensor_1.state = SENSOR_ERROR;
    diagnostics.gps_sensor_1.error_code = GPS_COMMUNICATION_FAILURE;
}
```

Architecture

Complete system overview showing FreeRTOS, sensor polling loop, protobuf encoding, UDP transmission, and memory layout. Includes initialization sequence and error handling strategy.

Main Loop Operation

```
while (1) {
    // STEP 1: System Health Check
    // - Check heap (critical: <8KB)
    // - Toggle status LEDs
    // - Send raw Ethernet beacon

    // STEP 2: Initialize Diagnostics Message
    SensorBoardDiagnostics diagnostics_msg;

    // STEP 3: Poll All Sensors
    // - pH Sensor
    // - GPS Sensor
    // - IMU Sensor
    // - Load Cells (x2)
    // - Pressure Sensors (x2)

    // STEP 4: Encode & Transmit
    // - Encode diagnostics to Protobuf
    // - Send UDP broadcast (192.168.0.255:7)

    // STEP 5: Wait
    osDelay(5000); // 5 seconds
}
```

Error Handling Strategy

Polling Error States

```
if (poll_result == RESULT_ERR_UNIMPLEMENTED) {
    sensor.state = SENSOR_IDLE;
    sensor.error_code = COMMUNICATION_FAILURE;

} else if (poll_result == RESULT_ERR_COMMS) {
    sensor.state = SENSOR_ERROR;
    sensor.error_code = COMMUNICATION_FAILURE;

} else if (poll_result == RESULT_OK) {
    if (validate_sensor_data(value) == RESULT_OK) {
        sensor.state = SENSOR_OPERATING;
        sensor.error_code = NO_ERROR;
    } else {
        sensor.state = SENSOR_ERROR;
        sensor.error_code = INVALID_DATA;
    }
}
```

Sensor Status Codes

Code	Meaning
`SENSOR_IDLE`	Not connected or not implemented
`SENSOR_OPERATING`	Normal operation, valid data
`SENSOR_ERROR`	Communication failure or invalid data

Initialization Sequence

Phase 1: Hardware Setup

```
1. MPU/Cache configuration
2. HAL initialization
3. System clock → 480 MHz
4. RTOS kernel init
5. GPIO initialization
6. Timer initialization (TIM1)
```

Phase 2: Communication Setup

1. UART initialization (115200 baud)
2. Logging system init
3. Ethernet PHY init (LAN8742)
4. MAC address filtering
5. ARP table setup

Phase 3: Application Setup

1. Packet dispatcher registration
2. Sensor initialization
3. UDP queue creation
4. UDP callback registration

Phase 4: Main Loop

1. LED initialization
2. Sensor polling starts
3. Continuous 5-second cycles

Configuration

Compile-time parameters, runtime configuration, sensor calibration setup, network addressing, UDP ports, performance tuning options

UDP Configuration

```
// In src/sensor_board/main.c
#define SENSOR_UDP_DEST_PORT 7
uint8_t dest_ip[4] = {192, 168, 0, 255};  // Broadcast address
```

Main Loop Timing

```
// In src/sensor_board/main.c
#define MAIN_TASK_DELAY_MS 5000             // 5 seconds between updates
```

Heap Management

```
// Critical heap threshold
#define CRITICAL_HEAP_THRESHOLD 8192        // 8 KB minimum
```

Task Configuration

```
const osThreadAttr_t mainTask_attributes = {
    .name = "mainTask",
    .stack_size = 1024 * 8,                 // 8 KB stack
    .priority = (osPriority_t)osPriorityNormal,
};
```

Runtime Configuration

Changing Sensor Poll Interval

```
// Current: 5 seconds
#define MAIN_TASK_DELAY_MS 5000

// Change to 1 second
```

```
#define MAIN_TASK_DELAY_MS 1000

// Change to 10 seconds
#define MAIN_TASK_DELAY_MS 10000
```

Enabling/Disabling Sensors

```
// Current: skip all sensor polling during development
const bool skip_sensor_polling = true;

// To ENABLE actual polling:
const bool skip_sensor_polling = false;

// To SKIP (for testing without hardware):
const bool skip_sensor_polling = true;
```

Adjusting Heap Threshold

```
// Current: 8 KB critical
if (free_heap < 8192) {
    LOGE(TAG, "CRITICAL: Low heap detected! Free: %lu bytes", free_heap);
}
```

Sensor Calibration Configuration

pH Sensor Calibration

```
// Set reference voltage (typically 3.3V or 5.0V)
ph_sensor_init(&ph_sensor, 3.3f);  // 3.3V reference

// Update calibration (after two-point calibration)
ph_sensor.calibration.offset = 0.0f;     // Adjust after pH 7.0 test
ph_sensor.calibration.slope = 3.5f;      // Adjust after second pH test
```

Load Cell Calibration

```
// After tare (no load):
load_cell_data[0].tare_offset_counts = adc_reading_no_load;

// After known weight measurement:
// scale = (force_newtons) / (adc_reading - tare_offset)
load_cell_data[0].scale_newtons_per_count = scale_factor;
load_cell_data[0].is_calibrated = true;
```

Network Configuration

Changing Broadcast Address

```
//To be implemented
```

Changing UDP Port

```
//To be implemented
```

GNSS

Global positioning module with NMEA 0183 protocol support. Contains data structures, initialization code, validation functions, and UART configuration for the GY-NEO6MV2 sensor.

Hardware Specifications

Parameter	Value
Model	GY-NEO6MV2 (NEO-6M)
Interface	UART (TTL serial)
Baud Rate	9600 bps
Protocol	NMEA 0183
Update Rate	1 Hz (configurable)

GPS Fix Quality Types

```
typedef enum {
    GPS_NO_FIX = 0,        // No GPS fix available
    GPS_GPS_FIX = 1,       // Standard GPS fix
    GPS_DGPS_FIX = 2,      // Differential GPS fix
    GPS_PPS_FIX = 3,       // PPS (Pulse Per Second) fix
    GPS_RTK_FIX = 4,       // Real-Time Kinematic fix
    GPS_RTK_FLOAT = 5      // RTK float solution
} gps_fix_quality_t;
```

Data Structure

```
typedef struct {
    // Position Data
    double latitude;           // ±90 degrees (North/South)
    double longitude;          // ±180 degrees (East/West)
    float altitude;            // Meters above sea level

    // Velocity & Direction
```

```
    float speed;               // Meters per second
    float heading;             // 0-360 degrees (course over ground)

    // Accuracy Indicators
    float hdop;                // Horizontal dilution of precision
    float vdop;                // Vertical dilution of precision
    int32_t satellites;        // Number of satellites in view
    gps_fix_quality_t fix_quality;

    // Timestamps
    int64_t utc_timestamp;     // Unix epoch (milliseconds)
    uint32_t timestamp;        // System timestamp (last reading)
    uint32_t last_timestamp;   // System timestamp (previous reading)

    // Status
    bool is_valid;             // Data validity flag

    // UART Parsing State
    char rx_buffer[256];       // UART receive buffer
    uint16_t rx_index;         // Current buffer position
    bool sentence_ready;       // Complete sentence available
} gps_data_t;
```

NMEA Sentences Supported

Sentence	Purpose
GGA	Fix data (time, position, fix quality, satellite count)
RMC	Recommended Min. Navigation Info (position, speed, heading, date)
GSA	DOP and active satellites (fix type, DOP values)

Initialization & Usage

Initialize GPS

```
gps_data_t gps_data;
gps_sensor_init(&gps_data);
```

Poll GPS Data

```
result_t gps_poll_result = poll_gps_sensor(&gps_data);

if (gps_poll_result == RESULT_OK) {
    double latitude = gps_data.latitude;
    double longitude = gps_data.longitude;
    float altitude = gps_data.altitude;
    int32_t satellites = gps_data.satellites;
    float hdop = gps_data.hdop;
}
```

Validation Functions

```
// Validate latitude (-90 to +90)
result_t validate_gps_latitude(double latitude);

// Validate longitude (-180 to +180)
result_t validate_gps_longitude(double longitude);

// Validate HDOP (horizontal dilution of precision)
// Typical range: 0-50
result_t validate_gps_hdop(float hdop);

// Validate satellite count
// Typical: 0-30 satellites
result_t validate_gps_satellite_count(int32_t satellites);
```

Protobuf Message Format

```
message SensorBoardGPSInfo {
    double latitude;
    double longitude;
    float altitude;
    float speed;
    float heading;
```

```
    float hdop;
    float vdop;
    int32 satellites;
    SensorState state;
    GPSErrorCode error_code;
}
```

Error Handling

```
if (gps_poll_result == RESULT_ERR_UNIMPLEMENTED) {
    // Hardware not connected
    diagnostics.gps_sensor_1.state = SensorState_SENSOR_IDLE;
    diagnostics.gps_sensor_1.error_code = GPSErrorCode_GPS_COMMUNICATION_FAILURE;
} else if (gps_poll_result == RESULT_ERR_COMMS) {
    // Communication error (timeout/CRC)
    diagnostics.gps_sensor_1.state = SensorState_SENSOR_ERROR;
} else if (gps_poll_result == RESULT_OK) {
    // Validate data before accepting
    if (validate_gps_latitude(gps_data.latitude) == RESULT_OK) {
        diagnostics.gps_sensor_1.state = SensorState_SENSOR_OPERATING;
    }
}
```

Integration Notes

- Configured for dual GPS redundancy capability
- UART buffer size: 256 bytes
- NMEA sentence max length: 83 characters
- Data published to network at main loop interval (5 seconds default)
- All GPS data transmitted via UDP with Protobuf encoding
- Temperature range: -40°C to +85°C (standard)

pH Sensor

The pH sensor provides water quality measurement critical for environmental monitoring and anomaly detection.

Hardware Specifications

Parameter	Value
Model	DFRobot SEN0161 (Analog pH meter)
Interface	Analog ADC
Reference Voltage	3.3V or 5.0V (configurable)
Output Range	0-5V analog
Measurement Range	0-14 pH units
Accuracy	±0.1 pH @ 25°C
Sample Rate	Configurable (40 samples for averaging)

Calibration Model

The sensor uses linear voltage-to-pH conversion:

```
pH = (Voltage / Reference_Voltage) × Slope + Offset
```

Default Parameters for SEN0161 @ 25°C:

- **Slope**: 3.5
- **Offset**: Variable (user calibration)

Data Structure

```
typedef struct {
    // Raw ADC Reading
    uint16_t raw_value;                // Raw ADC value
```

```
    // Calculated Values
    float voltage;                      // Converted voltage (0-5V)
    float ph_value;                     // Calculated pH (0-14)
    float reference_voltage;            // ADC reference (typically 3.3V or 5.0V)

    // Calibration Parameters
    ph_calibration_t calibration;       // { offset: float, slope: float }

    // Averaging Buffer (Noise Filtering)
    uint16_t sample_buffer[40];         // Last 40 samples
    uint8_t sample_index;               // Current position in buffer
    uint8_t samples_collected;          // Total samples collected (0-40)
} ph_sensor_t;
```

Initialization & Usage

Initialize pH Sensor

```
ph_sensor_t ph_sensor;
ph_sensor_init(&ph_sensor, 3.3f);  // 3.3V reference voltage
```

Poll pH Sensor

```
result_t ph_result = poll_ph_sensor(&ph_sensor);

if (ph_result == RESULT_OK) {
    float ph_value = ph_sensor.ph_value;
    float voltage = ph_sensor.voltage;
}
```

Manual Sample Addition

```
// For manual sampling at regular intervals
uint16_t adc_reading = 2048;  // Example ADC value
ph_sensor_add_sample(&ph_sensor, adc_reading);
```

Validation

```
result_t validate_ph_value(float ph_value);
// Returns RESULT_OK if 0 <= ph_value <= 14
// Returns RESULT_ERR_INVALID_DATA otherwise
```

Sample Averaging Strategy

Parameter	Value
Sample Buffer Size	40 samples
Method	Circular buffer moving average
Purpose	Noise filtering and stable readings
Typical Update Latency	40ms-800ms

Averaging Algorithm

```
1. ADC sample added to circular buffer
2. All 40 samples averaged together
3. Averaged value converted to voltage
4. Voltage converted to pH via calibration
```

Two-Point Calibration Procedure

Step 1: Neutral Point (pH 7.0)

```
1. Immerse electrode in pH 7.0 buffer solution
2. Wait for stable reading (~2 minutes)
3. Record voltage: V_neutral
4. Calculate offset adjustment
```

Step 2: Slope Calibration (pH 4.0 or 10.0)

```
1. Immerse electrode in second known pH solution
2. Wait for stable reading
3. Record voltage: V_reference
4. Calculate slope from two points:
   slope = (pH_reference - 7.0) / (V_reference - V_neutral)
```

Protobuf Message Format

```
message SensorBoardPHInfo {
    float ph_value;
    float voltage;
    SensorState state;
    PHErrorCode error_code;
}

enum PHErrorCode {
    PH_NO_ERROR = 0;
    PH_COMMUNICATION_FAILURE = 1;
    PH_INVALID_DATA = 2;
}
```

Error Handling

```
if (ph_result == RESULT_ERR_UNIMPLEMENTED) {
    // Hardware not connected
    diagnostics.ph_sensor.state = SensorState_SENSOR_IDLE;
    diagnostics.ph_sensor.error_code = PHErrorCode_PH_COMMUNICATION_FAILURE;
} else if (ph_result == RESULT_OK) {
    if (validate_ph_value(ph_sensor.ph_value) == RESULT_OK) {
        diagnostics.ph_sensor.state = SensorState_SENSOR_OPERATING;
        diagnostics.ph_sensor.error_code = PHErrorCode_PH_NO_ERROR;
    } else {
        // Invalid data from sensor (out of 0-14 range)
        diagnostics.ph_sensor.state = SensorState_SENSOR_ERROR;
        diagnostics.ph_sensor.error_code = PHErrorCode_PH_INVALID_DATA;
    }
```

```
}
```

Integration Notes

- Single sensor instance in main application
- Updates transmitted to network at main loop interval (5 seconds default)
- Temperature compensation not currently implemented (assumes ~25°C)
- Sample averaging reduces noise but introduces ~40ms latency per update
- Electrode response time: ~100-300ms depending on pH change magnitude

IMU

The Inertial Measurement Unit (IMU) provides three-axis acceleration, angular velocity, and magnetic field measurements for attitude determination and motion analysis.

Communication Interfaces

- **Accelerometer**: I2C or SPI
- **Gyroscope**: I2C or SPI
- **Magnetometer**: I2C or SPI
- **Update Rate**: Typically 1-100+ Hz (configurable)

Data Structure

```
typedef struct {
    // Acceleration (m/s² or g-units)
    float accel[3];            // [X, Y, Z] acceleration

    // Angular Velocity (rad/s or °/s)
    float gyro[3];             // [X, Y, Z] rotation rate

    // Magnetic Field (Gauss or µT)
    float mag[3];              // [X, Y, Z] magnetic field

    // Timestamps
    uint32_t timestamp;        // Current reading timestamp
    uint32_t last_timestamp;   // Previous reading timestamp
} imu_data_t;
```

Initialization & Usage

Initialize IMU

```
imu_data_t imu_data;
imu_sensor_init(&imu_data);
```

Poll IMU Data

```
result_t imu_result = poll_imu_sensor(&imu_data);

if (imu_result == RESULT_OK) {
    // Acceleration
    float accel_x = imu_data.accel[0];
    float accel_y = imu_data.accel[1];
    float accel_z = imu_data.accel[2];

    // Angular velocity
    float gyro_x = imu_data.gyro[0];
    float gyro_y = imu_data.gyro[1];
    float gyro_z = imu_data.gyro[2];

    // Magnetic field
    float mag_x = imu_data.mag[0];
    float mag_y = imu_data.mag[1];
    float mag_z = imu_data.mag[2];
}
```

Advanced Functions

Update with Raw Values

```
result_t imu_sensor_update(
    imu_data_t *imu,
    float ax, float ay, float az,  // Accelerometer values
    float gx, float gy, float gz,  // Gyroscope values
    float mx, float my, float mz,  // Magnetometer values
    uint32_t timestamp
);
```

Calculate Acceleration Magnitude

```
float acceleration_magnitude = imu_get_acceleration_magnitude(&imu_data);
// Useful for impact detection and free-fall detection
```

Thread-Safe Data Reading

```
imu_data_t imu_copy;
imu_sensor_read(&imu_data, &imu_copy);
// Use imu_copy in another thread without locking
```

Typical Sensor Ranges

Measurement	Typical Range	Units
Acceleration	±16	g
Angular Velocity	±2000	°/s
Magnetic Field	±4800	μT

Conversion Reference

From	To	Factor
g	m/s²	× 9.81
°/s	rad/s	× π/180
Gauss	μT	× 100

Validation Functions

```
// Validate single accelerometer value
// Typical range: -50 to +50 m/s2
result_t validate_accelerometer_value(float accel_value);

// Validate all three acceleration axes
result_t validate_imu_data(
```

```
    float accel_x,
    float accel_y,
    float accel_z
);
```

Protobuf Message Format

```
message SensorBoardIMUInfo {
    float accel_x;
    float accel_y;
    float accel_z;
    float gyro_x;
    float gyro_y;
    float gyro_z;
    SensorState state;
}
```

Common Applications

Impact Detection

```
float mag = imu_get_acceleration_magnitude(&imu_data);
if (mag > IMPACT_THRESHOLD) {
    // High acceleration detected
}
```

Tilt Detection

```
// Calculate tilt angle from acceleration
float tilt_angle = atan2(imu_data.accel[0], imu_data.accel[2]);
```

Motion Classification

```
// Static vs dynamic based on gyro magnitude
float gyro_mag = sqrt(gyro_x*gyro_x + gyro_y*gyro_y + gyro_z*gyro_z);
```

Integration Notes

- Single IMU instance with 3-axis sensors (dual planned)
- All three axes transmitted as independent fields
- Acceleration magnitude available for impact detection
- Timestamp tracking enables dead reckoning applications
- Typical IMU update rate: 10-100 Hz
- Filter algorithms can be applied to raw data for smoothing

Load Cell

Load cells measure force/weight to detect object presence, evaluate structural loading, or monitor mechanical stress. The system supports dual load cell configuration.

Hardware Specifications

Parameter	Value
Sensor Count	2 (independent)
Interface	Analog ADC
Measurement	Force (Newtons) / Mass (grams)
Update Rate	Configurable

Data Structure

```
typedef struct {
    // Raw Reading
    int32_t raw_counts;                // ADC value from sensor

    // Converted Measurements
    float force_newtons;               // Force in Newtons (N)
    float mass_grams;                  // Equivalent mass in grams (g)

    // Calibration Parameters
    float scale_newtons_per_count;     // Conversion factor (N/count)
    int32_t tare_offset_counts;        // Zero-load ADC offset

    // Status
    bool is_calibrated;                // Calibration valid flag
} load_cell_data_t;
```

Initialization

Initialize Load Cells

```
load_cell_data_t load_cell_data[2];  // Support 2 sensors

for (size_t i = 0; i < 2; i++) {
    load_cell_sensor_init(&load_cell_data[i]);
}
```

Poll Load Cell Sensor

```
result_t lc_result = poll_load_cell_sensor(&load_cell_data[0]);

if (lc_result == RESULT_OK) {
    float force = load_cell_data[0].force_newtons;
    float mass = load_cell_data[0].mass_grams;
    int32_t raw = load_cell_data[0].raw_counts;
}
```

Data Access Functions

```
// Get force in Newtons
result_t load_cell_get_force_newtons(
    const load_cell_data_t *data,
    float *force_newtons
);

// Get mass in grams (estimated)
result_t load_cell_get_mass_grams(
    const load_cell_data_t *data,
    float *mass_grams
);

// Get raw ADC counts
result_t load_cell_get_raw_counts(
    const load_cell_data_t *data,
    int32_t *raw_counts
);
```

```
// Get calibration parameters
result_t load_cell_get_calibration(
    const load_cell_data_t *data,
    float *scale_newtons_per_count,
    int32_t *tare_offset_counts
);

// Verify sensor validity
result_t load_cell_sensor_is_valid(
    const load_cell_data_t *data,
    bool *is_valid
);
```

Calibration Procedure

Two-Step Calibration

Step 1: Tare (Zero Load)

```
1. Remove all load from sensor
2. Measure ADC value: ADC_zero
3. Set tare_offset_counts = ADC_zero
```

Step 2: Span (Known Weight)

```
1. Place known weight on sensor
2. Measure ADC value: ADC_loaded
3. Know reference force: F_ref (Newtons)

4. Calculate scale:
   scale = (F_ref - 0.0) / (ADC_loaded - ADC_zero)

5. Set scale_newtons_per_count = scale
```

Measurement Formulas

```
// Raw force calculation
Force = (raw_counts - tare_offset_counts) × scale_newtons_per_count

// Mass conversion (approximate)
Mass_grams = (Force_newtons / 9.81) × 1000
           ≈ Force_newtons × 102.04
```

Dual Sensor Management

Configuration Example

```
// Initialize both sensors
for (size_t i = 0; i < 2; i++) {
    load_cell_sensor_init(&load_cell_data[i]);
}

// Poll both in sequence
poll_load_cell_sensor(&load_cell_data[0]);
poll_load_cell_sensor(&load_cell_data[1]);

// Access by index
float load_0 = load_cell_data[0].force_newtons;
float load_1 = load_cell_data[1].force_newtons;
```

Protobuf Message Format

```
message SensorBoardLoadCellInfo {
    uint32 sensor_index;         // 0 or 1
    float force_newtons;
    float mass_grams;
    SensorState state;
    LoadCellErrorCode error_code;
}
```

Unit Conversions

From	To	Factor
Newtons	kilograms-force (kgf)	÷ 9.81
Newtons	pounds-force (lbf)	÷ 4.448
grams	kilograms	÷ 1000

Manual Unit Conversion Example

```
// Convert to pounds-force
float force_lbf = load_cell_data[0].force_newtons / 4.448f;

// Convert to kilograms
float mass_kg = load_cell_data[0].mass_grams / 1000.0f;
```

Typical Specifications

Load Cell Type	Max Load	Accuracy
Strain gauge (±50N)	50N	±0.1%
Load cell (±100N)	100N	±0.05%
Heavy duty (±1000N)	1000N	±0.1%

Integration Notes

- Supports up to 2 independent load cell sensors
- Hardware-specific ADC implementation
- Force conversion via linear scaling model
- Tare offset corrects for sensor mechanical zero
- Real-time monitoring of structural loads
- Each sensor maintains separate calibration
- Independent error reporting per sensor

Pressure Sensor

Pressure sensors measure fluid/gas pressure for robotic gripper force feedback and control, depth sensing, altitude measurement, or system pressure monitoring. The system supports dual pressure sensor configuration for dual-pad gripper control with load distribution feedback.

Hardware Specifications

Parameter	Value
Sensor Count	2 (independent)
Interface	Analog ADC or I2C
Measurement Range	Variable (typically 0-300 kPa)
Update Rate	Configurable

Data Structure

```
typedef struct {
    float pressure_kpa;         // Pressure in kilopascals (kPa)
    float temperature_c;        // Temperature in Celsius (°C)
    float voltage;              // Sensor output voltage
    bool is_calibrated;         // Calibration status flag
} pressure_sensor_data_t;
```

Initialization

Initialize Pressure Sensors

```
pressure_sensor_data_t pressure_data[2];  // Support 2 sensors

for (size_t i = 0; i < 2; i++) {
    pressure_sensor_init(&pressure_data[i]);
}
```

Poll Pressure Sensor

```
result_t ps_result = poll_pressure_sensor(&pressure_data[0]);

if (ps_result == RESULT_OK) {
    float pressure_kpa = pressure_data[0].pressure_kpa;
    float temperature_c = pressure_data[0].temperature_c;
}
```

Data Access Functions

```
// Get pressure in kilopascals
result_t pressure_sensor_get_pressure_kpa(
    const pressure_sensor_data_t *data,
    float *pressure_kpa
);

// Get temperature in Celsius
result_t pressure_sensor_get_temperature_c(
    const pressure_sensor_data_t *data,
    float *temperature_c
);

// Get raw sensor voltage
result_t pressure_sensor_get_voltage(
    const pressure_sensor_data_t *data,
    float *voltage
);

// Verify sensor validity
result_t pressure_sensor_is_valid(
    const pressure_sensor_data_t *data,
    bool *is_valid
);
```

Pressure Unit Conversions

Function-Based Conversions

```
// Convert pressure from bar to psi
result_t bar_to_psi(float bar, float *psi);

// Convert pressure from psi to bar
result_t psi_to_bar(float psi, float *bar);
```

Conversion Table

From	To	Multiply By
bar	kPa	100
psi	kPa	6.895
atm	kPa	101.325
kPa	bar	0.01
kPa	psi	0.145
kPa	atm	0.00987

Examples

```
// 100 kPa = 1 bar
float kpa = 100.0f;
float bar = kpa * 0.01f;  // Result: 1.0 bar

// 50 psi to bar
float psi = 50.0f;
float bar = psi / 14.504f;  // Result: 3.45 bar

// Altitude from pressure (simplified)
// Altitude ≈ 44330 × (1 - (P/P0)^(1/5.255))
float altitude_m = 44330.0f * (1.0f - pow(pressure_kpa/101.325f, 1.0f/5.255f));
```

Protobuf Message Format

```
message SensorBoardPressureInfo {
    uint32 sensor_index;         // 0 or 1
    float pressure_kpa;
    float temperature_c;
    SensorState state;
    PressureErrorCode error_code;
}
```

Dual Sensor Management

Configuration Example

```
// Initialize both sensors
for (size_t i = 0; i < 2; i++) {
    pressure_sensor_init(&pressure_data[i]);
}

// Poll both in sequence
poll_pressure_sensor(&pressure_data[0]);
poll_pressure_sensor(&pressure_data[1]);

// Access by index
float pressure_0_kpa = pressure_data[0].pressure_kpa;
float pressure_1_kpa = pressure_data[1].pressure_kpa;
```

Temperature Compensation

Pressure readings often need temperature compensation for accuracy:

```
// Simplified temperature compensation
float compensated_pressure = pressure_data[0].pressure_kpa *
    (reference_temperature + 273.15f) /
    (pressure_data[0].temperature_c + 273.15f);
```

Applications

Robotic Gripper Control (Primary Use Case)

```
// Gripper force feedback for adaptive grip strength
// Pressure reading controls servo/motor PWM to regulate grip force

#define GRIPPER_MIN_PRESSURE_KPA 20.0f   // Minimum safe grip
#define GRIPPER_MAX_PRESSURE_KPA 150.0f  // Maximum allowed grip
#define GRIPPER_TARGET_PRESSURE_KPA 80.0f // Desired grip force

// PID controller for gripper force regulation
typedef struct {
    float kp, ki, kd;            // PID coefficients
    float integral_error;
    float previous_error;
} gripper_pid_t;

// Adjust servo PWM based on pressure feedback
void adjust_gripper_force(float current_pressure_kpa, gripper_pid_t *pid) {
    float error = GRIPPER_TARGET_PRESSURE_KPA - current_pressure_kpa;

    pid->integral_error += error;
    float derivative_error = error - pid->previous_error;

    float pid_output = (pid->kp * error) +
                       (pid->ki * pid->integral_error) +
                       (pid->kd * derivative_error);

    // Clamp servo PWM to valid range
    uint16_t servo_pwm = (uint16_t)(GRIPPER_NEUTRAL_PWM + pid_output);
    servo_pwm = (servo_pwm < GRIPPER_MIN_PWM) ? GRIPPER_MIN_PWM : servo_pwm;
    servo_pwm = (servo_pwm > GRIPPER_MAX_PWM) ? GRIPPER_MAX_PWM : servo_pwm;

    set_gripper_pwm(servo_pwm);
    pid->previous_error = error;
}
```

Gripper Control Features:

- Grip force feedback for object handling
- Object presence detection (pressure spike threshold)

- Adaptive compliance for varying object sizes/materials
- Dual sensors support load sharing across gripper pads

Implemented but Not Primary

Depth Sensing (Water)

```
// Pressure to depth in water
// P = ρ × g × h
// where ρ = 1025 kg/m³ (seawater), g = 9.81 m/s²
float depth_meters = (pressure_kpa - atmospheric_pressure_kpa) / 10.0f;
```

Altitude Sensing (Air)

```
// Barometric formula (simplified)
float altitude_m = 44330.0f * (1.0f - pow(pressure_kpa/101.325f, 1.0f/5.255f));
```

System Pressure Monitoring

```
if (pressure_kpa > PRESSURE_WARNING_THRESHOLD) {
    // High pressure detected - safety alert
}
```

Common Pressure Sensor Ranges

Application	Range	Typical Sensor
Altitude (aviation)	10-110 kPa	BMP280/BMP390
Depth (diving)	0-300+ kPa	Custom depth sensor
System pressure	0-500+ kPa	Industrial pressure transducer

Integration Notes

- **Primary Application:** Robotic gripper force feedback and control
- Supports up to 2 independent pressure sensors (dual gripper pads)
- Temperature measurement for compensation algorithms
- Hardware-specific ADC or I2C implementation
- Pressure-voltage conversion implemented internally
- Real-time depth/altitude sensing capability

- PID control loop integration for adaptive grip force
- Each sensor maintains independent calibration
- Temperature tracking for accuracy improvements
- Slip detection via pressure variance analysis

Testing

Test organization, validation functions, Unity testing framework usage, manual hardware testing checklists and debugging/troubleshooting guide.

Test Organization

```
test/sensor_board/
├── test_gps_sensor/
│   ├── NMEA parsing verification
│   ├── Coordinate validation
│   └── Fix quality enumeration tests
│
├── test_imu_sensor/
│   ├── 3-axis data structure tests
│   ├── Acceleration magnitude calculation
│   └── Timestamp tracking validation
│
├── test_load_sensor/
│   ├── Force calculation (tare/scale)
│   ├── Mass conversion (g/kg/lbf)
│   ├── Dual sensor independence
│   └── Calibration parameter storage
│
├── test_ph_sensor/
│   ├── ADC to voltage conversion
│   ├── pH value calculation
│   ├── Sample averaging (40-sample buffer)
│   ├── Calibration offset/slope
│   └── Edge cases (pH 0, 14)
│
├── test_pressure_sensor/
│   ├── Pressure reading validation
│   ├── Temperature compensation
│   ├── Unit conversions (bar/psi/kPa)
│   └── Dual sensor independence
```

```
│
└── test_sensor_basics/
    ├── Conversion functions
    ├── Validation functions
    ├── Range checking
    └── Boundary conditions
```

Validation Functions

GPS Validation

```
result_t validate_gps_latitude(double latitude);
// Valid range: -90.0 to +90.0 degrees

result_t validate_gps_longitude(double longitude);
// Valid range: -180.0 to +180.0 degrees

result_t validate_gps_hdop(float hdop);
// Typical range: 0.0 to 50.0

result_t validate_gps_satellite_count(int32_t satellites);
// Typical range: 0 to 30 satellites
```

pH Validation

```
result_t validate_ph_value(float ph_value);
// Valid range: 0.0 to 14.0
```

IMU Validation

```
result_t validate_accelerometer_value(float accel_value);
// Typical range: -50.0 to +50.0 m/s²

result_t validate_imu_data(float accel_x, float accel_y, float accel_z);
```

Unit Conversion Tests(extra stuff)

```
result_t celsius_to_fahrenheit(float celsius, float *fahrenheit);
result_t fahrenheit_to_celsius(float fahrenheit, float *celsius);
result_t bar_to_psi(float bar, float *psi);
result_t psi_to_bar(float psi, float *bar);
```

Test Framework

Testing Technology

- **Framework**: Unity (open-source testing framework)
- **Build System**: CMake / PlatformIO
- **Test Type**: Hosted unit tests (run on PC)
- **Mocking**: Mock ADC/UART/I2C interfaces

Building Tests

```
// Build all tests
pio test -e sensor_board

// Run specific test suite
pio test -e sensor_board -f test_ph_sensor
```

Debugging & Troubleshooting

Issue	Cause	Solution
Sensor IDLE	Not connected	Check UART/I2C/SPI wiring
Sensor ERROR	Communication failure	Verify baud rates, addresses
Invalid data	Out of range	Check calibration parameters
Low heap warning	Memory leak	Review packet encoder

Reference

Checkout the embedded simplified from ERC

Source Code References

If you made it till here, you a true G. Also, this was rather useful, check this out... ERC Embedded Simplified Official

Main Application

File	Purpose
`src/sensor_board/main.c`	Main entry point and sensor loop

Sensor Drivers

Sensor	Header	Source
GPS	`components/sensor_board/gps/gps_sensor.h`	`gps_sensor.c`
IMU	`components/sensor_board/imu/imu_sensor.h`	`imu_sensor.c`
pH	`components/sensor_board/ph/ph_sensor.h`	`ph_sensor.c`
Load Cell	`components/sensor_board/load_cell/load_cell_sensor.h`	`load_cell_sensor.c`
Pressure	`components/sensor_board/pressure/pressure_sensor.h`	`pressure_sensor.c`
Utilities	`components/sensor_board/sensor_basics/sensor_basics.h`	`sensor_basics.c`

Protobuf Definitions

For detailed protobuf message format documentation, see: Sensor Board Protobuf

```
ERC-Protobufs/components/sensor_board/
├── diagnostics.proto       - Board-level diagnostics and health status
├── sensor.proto            - Aggregated sensor state message
```

```
├── gps_sensor.proto        - GNSS positioning data
├── imu_sensor.proto        - Inertial measurement (acceleration, gyro, mag)
├── ph_sensor.proto         - Water quality pH measurement
├── load_cell.proto         - Force measurement (gripper control)
└── pressure_sensor.proto   - Pressure measurement (gripper control)
```

Build Configuration

File	Purpose
`platformio.ini`	Build configuration for all boards
`.mxproject`	STM32 CubeMX configuration
`firmware.ioc`	CubeMX IOC file (device config)

Test References

Test Suites

```
test/sensor_board/
├── test_gps_sensor/
├── test_imu_sensor/
├── test_load_sensor/
├── test_ph_sensor/
├── test_pressure_sensor/
└── test_sensor_basics/
```

Running Tests

```
// Build all tests
pio test -e sensor_board

// Run with verbose output
pio test -e sensor_board -v

// Build only (no run)
pio test -e sensor_board --no-run
```

Hardware References

Microcontroller

- **STM32H753ZI** - ARM Cortex-M7, 480 MHz, 1 MB Flash
- **Datasheet**: ST Microelectronics STM32H7 reference

Ethernet

- **LAN8742** - Ethernet PHY
- **Interface**: RMII (Reduced Media Independent Interface)
- **Link Speed**: 10/100 Mbps auto-negotiation

Sensors

Sensor	Model	Protocol	Note
GPS	GY-NEO6MV2 (NEO-6M)	UART 9600	Default NMEA config
IMU	Reference design TBD	I2C/SPI	Dual IMU planned
pH	DFRobot SEN0161	ADC Analog	40-sample averaging
Load Cell	Application specific	ADC	Dual cells (×2)
Pressure	Application specific	ADC/I2C	Dual sensors (×2)

Development Workflow

Building Firmware

```
// Build for sensor_board
pio run -e sensor_board

// Upload to board
pio run -e sensor_board --target upload

// Monitor serial output
pio device monitor -b 115200 -p COM4
```

Development Cycle

```
// 1. Edit source code
// 2. Build
pio run -e sensor_board

// 3. Upload
pio run -e sensor_board --target upload

// 4. Monitor
pio device monitor

// 5. Run tests
pio test -e sensor_board
```

Useful Commands

PlatformIO CLI

```
// List available boards
pio boards

// Show board info
pio boards nucleo_h753zi

// Clean build
pio run -e sensor_board --target clean

// Full rebuild
pio run -e sensor_board --target clean --target upload
```

Troubleshooting Guide

Serial Monitor No Output

1. Check USB connection

2. Verify COM port in platformio.ini
3. Check baud rate (115200)
4. Verify UART initialization succeeded

Sensor Shows IDLE

1. Check physical connection (UART/I2C/SPI)
2. Verify baud rate (for UART sensors)
3. Check pull-up resistors (for I2C)
4. Verify device address (for I2C/SPI)

Network Packets Not Received

1. Check IP address configuration
2. Verify MAC address filtering
3. Check firewall settings
4. Verify UDP port 7 not blocked

Low Heap Warning

1. Check for memory leaks in sensor drivers
2. Reduce buffer sizes
3. Verify UDP queue sizes
4. Check for recursive allocations

Quick Reference Checklist

Before Deployment

- ☐ All sensors connected and responding
- ☐ Network IP/MAC configured correctly
- ☐ Calibration parameters set for pH/load cells
- ☐ Serial monitor showing sensor data
- ☐ Heap usage monitored (>8KB free)
- ☐ UDP packets reaching destination
- ☐ Protobuf encoding verified

Monitoring in Production

- [ ] Check sensor status codes
- [ ] Monitor heap usage trend
- [ ] Verify data ranges match expectations
- [ ] Track error rates per sensor
- [ ] Review network packet statistics

Hope you had fun☮

END OF DOCUMENTATION FOR SESOR BOARD☮

Last Updated: April 14, 2026