Protobuffers

Starting with Protobufs

What are protobufs?

Protocol Buffers (protobufs) are a way to define structured messages in `.proto` files.

The point is simple: define the message format once, generate code for your language, and now everyone agrees what the bytes mean (without inventing a new packet format every semester).

Where they live

The source of truth for RoboTeam message definitions is: RoboTeamTwente/ERC-Protobufs.

Other repositories (embedded, software, tooling) should **consume** these definitions (often via a git submodule) instead of copy-pasting `.proto` files around.

Editing rules (the important part)

Field numbers are the real API

In protobuf, the **field number** is what goes on the wire. Renaming a field is usually harmless; changing its number is usually the problem.

Never renumber an existing field unless you are intentionally breaking compatibility and coordinating the update across all consumers.

Safe changes (usually)

- Add a new field with a **new** unused number.
- Add new enum values (don't reuse old numeric values for new meanings).
- Stop using a field before deleting it, and reserve its number if your style guide does that.

Changes that need extra care

- Changing a field type (e.g. `int32` → `string`).

- Changing semantics/units (mm vs m, degrees vs radians).

Typical workflow

1. Edit / add `.proto` files in `ERC-Protobufs`.
2. Generate / update code as required by your project (this step is repo-specific).
3. Update consumers to use the new fields (and handle missing fields safely).
4. Test at least one real send/receive path.
5. Open a PR that lists: what changed, field numbers, and compatibility expectations.

Troubleshooting

“My message doesn’t parse”

- Confirm both sides use the same (or compatible) `.proto` definitions.
- Check field numbers: no reuse, no renumbering.
- Check transport framing (length prefix / delimiter / CRC). Protobuf gives bytes; transport still matters.

“It parses, but values are nonsense”

- Check units and semantics (the most common “it’s wrong but not broken” bug).
- Make sure generated code is up to date (stale generated files cause creative failures).

> Protobufs are boring on purpose. Boring schemas prevent exciting debugging sessions. Keep changes small, reviewable, and compatible.

Design

This page explains rules that apply to all protobuffers, and a bit on how we use them.

PBEnvelope

All protobuffers are wrapped in the `PBEnvelope` message, which contains metadata we need in all protobufs.

Lost packets & throttling

In every line of communications, the packets must be sent periodically. The frequency of sent packets is indicated in their `PBEnvelope`. If the receiver does not get a packet within given timeout, it means the packet was lost. In most cases this means that the brakes should be engaged for safety.

Arm Board Protobuffers

This page: *each arm_board protobuffer explained.*

> The protobuffers for the arm_board are passed between software, control and embedded.

Movement_software_target.proto

Software -> control

This file contains two protobuffers detailing the source of a movement.

ArmBoardTargetMovement contains the target coordinates for a movement. This gets sent to control, they use this information to calculate motor angles.

ArmBoardObstructions is a placeholder for now. The idea is that software would be able to pass a list of coordinates of physical obstructions to control, around which they would manoeuvre the movement of the arm.

Movement_control_in.pb.h

Control -> Embedded

After control calculates the angles using the information from the previous protobuf, it sends the control signals back to us (embedded) with the absolute angles and frequency for the motors. So, this is the information embedded uses to actuate the motors.

ArmboardControlSignals contains all the angles the motors need to be turned to. We then use this information to set the PWM pins.

Movement_software_feedback.proto

Embedded -> Software

After the movement happens (or fails), embedded will send feedback to software for them to calculate the next movement.

ArmBoardMovementFeedback sends an error code to software.

Possible error codes are:

- Point_not_in_range
- Obstruction
- Calibration
- Motor_malfunction
- All_ok

ArmBoardActualPositions sends back the angle of each motor of the arm. Software will use this to display a 3D model of the arm position. (Allegedly)

Motor_diagnostics.proto

Arm board -> Debugging board

This protobuf gets periodically sent to the debugging board to indicate the status.

ArmBoardDiagnostics the `MotorInformation` for each motor and the state of the **entire** board.

Possible states are:

- Idle
- Operating
- Calibrating
- Errored

> **MotorInformation** is a common protobuf shared between all boards that use motors (this and driving board). MotorInformation contains the state (same as above), motor_id, rpm, voltage and encoder_angle for a single motor.

Sensor Board Protobuf

This page: *Complete protobuf message definitions for the sensor_board component.*

> The protobuffers for the sensor_board are passed between embedded, the network, and other boards for diagnostics and data collection. This page documents the actual message definitions from the ERC-Protobufs repository.

Common Definitions (sensor.proto)

Location: ERC-Protobufs/components/sensor_board/sensor.proto

SensorState Enum

Shared across all sensor types - indicates the current operational state of a sensor.

```
enum SensorState {
  SENSOR_IDLE = 0;           // Sensor is idle/not currently sampling
  SENSOR_OPERATING = 1;      // Sensor is actively sampling/operating normally
  SENSOR_CALIBRATING = 2;    // Sensor is in calibration mode
  SENSOR_ERROR = 3;          // Sensor has encountered an error
}
```

GPS Sensor (gps_sensor.proto)

Location: ERC-Protobufs/components/sensor_board/gps_sensor.proto

Direction: Sensor Board → Navigation/Localization Systems

GPSFixQuality Enum

Indicates the type and quality of GPS fix obtained.

```
enum GPSFixQuality {
  NO_FIX = 0;       // No fix available
  GPS_FIX = 1;      // Standard GPS fix
  DGPS_FIX = 2;     // Differential GPS fix
  PPS_FIX = 3;      // PPS (Pulse Per Second) fix
  RTK_FIX = 4;      // Real-Time Kinematic fix
  RTK_FLOAT = 5;    // RTK float solution
}
```

GPSErrorCode Enum

Detailed error codes for GPS/GNSS sensor failures.

```
enum GPSErrorCode {
  GPS_NO_ERROR = 0;
  GPS_COMMUNICATION_FAILURE = 1;   // UART connection lost
  GPS_INVALID_DATA = 2;             // Data parsing or validation failed
  GPS_ANTENNA_FAULT = 3;            // GPS antenna disconnected or faulty
  GPS_LOW_SIGNAL_QUALITY = 4;       // Signal strength too weak for fix
}
```

SensorBoardGPSInfo Message

Complete GPS positioning and velocity information.

```
message SensorBoardGPSInfo {
  // GPS coordinates
  double latitude = 1;           // Degrees (positive = North, negative = South)
  double longitude = 2;          // Degrees (positive = East, negative = West)
  float altitude = 3;            // Meters above sea level

  // Velocity data
  float speed = 4;               // Speed in meters per second
  float heading = 5;             // Course over ground in degrees (0-360)

  // Position accuracy and quality
  float hdop = 6;                // Horizontal dilution of precision (lower is better)
  float vdop = 7;                // Vertical dilution of precision (lower is better)
  int32 satellites = 8;          // Number of satellites in view (up to 16)
```

```
  GPSFixQuality fix_quality = 9;

  SensorState state = 10;
  GPSErrorCode error_code = 11;  // Error code if state is SENSOR_ERROR

  // Timestamp from GPS
  int64 utc_timestamp = 12;      // Unix timestamp in milliseconds
}
```

IMU Sensor (imu_sensor.proto)

Location: ERC-Protobufs/components/sensor_board/imu_sensor.proto

Direction: Sensor Board → Motion Control/Attitude Systems

IMUErrorCode Enum

Detailed error codes for IMU sensor failures.

```
enum IMUErrorCode {
  IMU_NO_ERROR = 0;
  IMU_COMMUNICATION_FAILURE = 1;      // I2C/SPI connection lost
  IMU_CALIBRATION_REQUIRED = 2;       // Sensor needs calibration
  IMU_CALIBRATION_FAILED = 3;         // Calibration procedure failed
  IMU_INVALID_DATA = 4;               // Data out of valid range
  IMU_SENSOR_FAULT = 5;               // Hardware fault detected
  IMU_GYROSCOPE_ERROR = 6;            // Gyroscope component failure
  IMU_MAGNETOMETER_ERROR = 7;         // Magnetometer component failure
  IMU_ACCELEROMETER_ERROR = 8;        // Accelerometer component failure
}
```

SensorBoardIMUInfo Message

3-axis inertial measurement data (acceleration, angular velocity, magnetic field).

```
message SensorBoardIMUInfo {
  // 3-Axis Accelerometer data
```

```
  float accel_x = 1;             // X-axis acceleration
  float accel_y = 2;             // Y-axis acceleration
  float accel_z = 3;             // Z-axis acceleration

  // 3-Axis Gyroscope data
  float gyro_x = 4;              // X-axis angular velocity
  float gyro_y = 5;              // Y-axis angular velocity
  float gyro_z = 6;              // Z-axis angular velocity

  // 3-Axis Magnetometer data (9-axis IMU only)
  float mag_x = 7;               // X-axis magnetic field
  float mag_y = 8;               // Y-axis magnetic field
  float mag_z = 9;               // Z-axis magnetic field

  bool is_calibrated = 13;       // True if IMU has been calibrated on level surface

  SensorState state = 14;
  IMUErrorCode error_code = 15;  // Error code if state is SENSOR_ERROR
}
```

---

pH Sensor (ph_sensor.proto)

Location: ERC-Protobufs/components/sensor_board/ph_sensor.proto

Direction: Sensor Board → Environmental Monitoring Systems

PHErrorCode Enum

Detailed error codes for pH sensor failures.

```
enum PHErrorCode {
  PH_NO_ERROR = 0;
  PH_COMMUNICATION_FAILURE = 1;       // ADC connection lost
  PH_OUT_OF_RANGE = 2;                // Reading outside 0-14 range
  PH_CALIBRATION_REQUIRED = 3;        // Sensor needs calibration
  PH_INVALID_DATA = 4;                // Data validation failed
  PH_PROBE_FAULT = 5;                 // Electrode fault detected
  PH_TEMPERATURE_SENSOR_ERROR = 6;    // Temperature compensation sensor failed
```

```
}
```

SensorBoardPHInfo Message

Water quality measurement with temperature compensation.

```
message SensorBoardPHInfo {
  float ph_value = 1;            // Raw pH measurement (0-14 scale, 7 is neutral)
  float voltage = 2;             // Voltage reading from pH sensor in millivolts
  float temperature = 3;         // Temperature reading in Celsius (if external sensor
available)

  SensorState state = 4;         // Current state of the pH sensor
  PHErrorCode error_code = 5;    // Error code if state is SENSOR_ERROR
}
```

Notes:

- pH value is derived from voltage via linear calibration equation
- 40-sample moving average applied internally to reduce ADC noise
- Temperature used for automatic compensation (default ~25°C)

Load Cell Sensor (load_cell.proto)

Location: ERC-Protobufs/components/sensor_board/load_cell.proto

Direction: Sensor Board → Arm Control/Gripper Systems

LoadCellErrorCode Enum

Detailed error codes for load cell sensor failures.

```
enum LoadCellErrorCode {
  LOAD_CELL_NO_ERROR = 0;
  LOAD_CELL_COMMUNICATION_FAILURE = 1;   // ADC connection lost
  LOAD_CELL_OUT_OF_RANGE = 2;            // Force measurement exceeds limits
  LOAD_CELL_CALIBRATION_REQUIRED = 3;    // Sensor needs calibration
  LOAD_CELL_INVALID_DATA = 4;            // Data validation failed
```

```
  LOAD_CELL_SENSOR_FAULT = 5;           // Hardware fault detected
}
```

SensorBoardLoadCellInfo Message

Force measurement for robotic gripper control and load monitoring.

```
message SensorBoardLoadCellInfo {
  uint32 sensor_index = 1;              // 0-based index of the load cell (0 or 1)
  float force_newtons = 2;              // Force in Newtons
  float mass_grams = 3;                 // Mass in grams (derived from force)
  int32 raw_counts = 4;                 // Raw ADC counts (for debugging)

  // Calibration status and parameters
  bool is_calibrated = 5;               // True if calibration parameters are valid
  float scale_newtons_per_count = 6;    // Conversion factor (N/count)
  int32 tare_offset_counts = 7;         // Zero-load ADC offset

  SensorState state = 8;
  LoadCellErrorCode error_code = 9;     // Error code if state is SENSOR_ERROR
}
```

Dual Sensor Support:

- Up to 2 independent load cells (sensor_index 0 and 1)
- Typically used on dual-pad grippers for load distribution feedback
- Each sensor maintains independent calibration parameters
- Enables slip detection via load imbalance analysis

Pressure Sensor (pressure_sensor.proto)

Location: ERC-Protobufs/components/sensor_board/pressure_sensor.proto

Direction: Sensor Board → Arm Control/Gripper Systems / Environmental Monitoring

PressureErrorCode Enum

Detailed error codes for pressure sensor failures.

```
enum PressureErrorCode {
  PRESSURE_NO_ERROR = 0;
  PRESSURE_COMMUNICATION_FAILURE = 1;    // I2C/ADC connection lost
  PRESSURE_OUT_OF_RANGE = 2;             // Reading exceeds sensor limits
  PRESSURE_CALIBRATION_REQUIRED = 3;     // Sensor needs calibration
  PRESSURE_INVALID_DATA = 4;             // Data validation failed
  PRESSURE_SENSOR_FAULT = 5;             // Hardware fault detected
}
```

SensorBoardPressureInfo Message

Pressure measurement for robotic gripper control and environmental sensing.

```
message SensorBoardPressureInfo {
  uint32 sensor_index = 1;               // 0-based index of the pressure sensor (0 or 1)

  // Pressure data
  float pressure_kpa = 2;                // Pressure in kilopascals
  float temperature_c = 3;               // Temperature in Celsius (if available)
  float voltage = 4;                     // Sensor output voltage (if available)

  // Calibration status
  bool is_calibrated = 5;                // True if calibration parameters are valid

  SensorState state = 6;
  PressureErrorCode error_code = 7;      // Error code if state is SENSOR_ERROR
}
```

Dual Sensor Support:

- Up to 2 independent pressure sensors (sensor_index 0 and 1)
- Primary use: gripper pad force feedback for PID control
- Enables real-time grip force regulation and slip detection
- Secondary uses: depth sensing, altitude sensing, system pressure monitoring

---

Board Diagnostics (diagnostics.proto)

Location: ERC-Protobufs/components/sensor_board/diagnostics.proto

Direction: Sensor Board → Debugging Board / Health Monitoring Systems

SensorBoardDiagnostics Message

Complete system health and status snapshot sent periodically (5 second default).

```
message SensorBoardDiagnostics {
  enum State {
    IDLE = 0;
    OPERATING = 1;
    CALIBRATING = 2;
    ERROR = 3;
  }

  State state = 1;                      // Overall board state
  SensorBoardPHInfo ph_sensor = 2;      // pH sensor status
  SensorBoardIMUInfo imu_sensor = 3;    // IMU sensor status

  // Overall sensor board health
  float board_temperature = 4;          // Board temperature in Celsius
  float board_voltage = 5;              // System voltage (3.3V supply)

  SensorBoardGPSInfo gps_sensor_1 = 6;  // GPS sensor status
}
```

Board States:

- **IDLE (0)** - Board initialized but not actively polling sensors
- **OPERATING (1)** - All sensors functioning normally, data being collected
- **CALIBRATING (2)** - Board or sensors in calibration mode
- **ERROR (3)** - Critical system failure detected

Health Indicators:

- **board_temperature** - STM32H753 die temperature; normal range 25-75°C
- **board_voltage** - 3.3V supply input; should be 3.0-3.6V
- **Embedded sensor states** - Each sensor's current SensorState (IDLE/OPERATING/CALIBRATING/ERROR)

Error Code Pattern

All sensor error codes follow a consistent pattern:

```
enum XXXErrorCode {
  XXX_NO_ERROR = 0;                     // No error
  XXX_COMMUNICATION_FAILURE = 1;        // Hardware interface (I2C/SPI/UART/ADC) failure
  XXX_OUT_OF_RANGE = 2;                 // Reading outside valid sensor range
  XXX_CALIBRATION_REQUIRED = 3;         // Sensor needs calibration
  XXX_INVALID_DATA = 4;                 // Data validation/filtering failed
  XXX_SENSOR_FAULT = 5;                 // Hardware fault or component failure
  // Additional sensor-specific errors...
}
```

Integration into application:

```
if (gps_result.error_code == GPS_COMMUNICATION_FAILURE) {
    LOG_ERROR("GPS UART connection lost");
    diagnostics.gps_sensor_1.state = SENSOR_ERROR;
} else if (gps_result.error_code == GPS_NO_ERROR) {
    LOG_INFO("GPS position: %f, %f", gps_result.latitude, gps_result.longitude);
}
```

Network Transmission

Main Sensor Data:

- Sent via individual sensor messages
- Packaged into SensorBoardState (sensor.proto)
- UDP broadcast to 192.168.0.255:7 (configurable)
- Interval: 5 seconds default (configurable)
- Encoding: Nanopb (lightweight protobuf for embedded)

Diagnostics Data:

- Sent via SensorBoardDiagnostics message
- UDP broadcast to 192.168.0.255:7 (same port)
- Interval: 5 seconds (synced with main loop)
- Contains snapshot of all sensor states + health metrics

Message Composition

The SensorBoardDiagnostics message is composed of individual sensor message types:

```
SensorBoardDiagnostics
├── state (Board state)
├── SensorBoardPHInfo
│   ├── ph_value
│   ├── voltage
│   ├── temperature
│   ├── state (SensorState)
│   └── error_code (PHErrorCode)
├── SensorBoardIMUInfo
│   ├── accel_x, accel_y, accel_z
│   ├── gyro_x, gyro_y, gyro_z
│   ├── mag_x, mag_y, mag_z
│   ├── is_calibrated
│   ├── state (SensorState)
│   └── error_code (IMUErrorCode)
├── board_temperature
├── board_voltage
└── SensorBoardGPSInfo
    ├── latitude, longitude, altitude
    ├── speed, heading
    ├── hdop, vdop, satellites
    ├── fix_quality
    ├── state (SensorState)
    ├── error_code (GPSErrorCode)
    └── utc_timestamp
```

Note: Not all proto files define load cells and pressure sensors in diagnostics.proto yet - they are sent as separate messages when available.

Driving Board Protobuffers

This page: *each driving_board protobuffer explained.*

> The protobuffers for the driving_board are passed between software, control and embedded.

DrivingBoardDiagnostics.proto

Embedded → Software / Debugging

This file contains the full diagnostic state of the driving board and all attached motors.

DrivingBoardDiagnostics is the main status message used to report the system state and motor-level health information.

DrivingBoardDiagnostics

This message contains:

- Overall board state (IDLE, OPERATING, CALIBRATING, ERROR)
- Motor information for all driving and steering motors

The board state is used to indicate the current operating mode of the driving board.

MotorInformation fields:

Each motor is reported using the MotorInformation protobuf, which includes:

- motor state
- motor ID
- RPM
- voltage
- encoder angle

Motors included The diagnostics message explicitly contains 10 motors: front_left_motor middle_left_motor back_left_motor front_right_motor middle_right_motor back_right_motor steering_front_left_motor steering_back_left_motor steering_front_right_motor steering_back_right_motor

>

> Notes: This structure is currently fixed-size, meaning all motors are explicitly defined instead of using a repeated field.

DrivingBoardMotorMessage.proto

Software → Embedded

This message is used to send motion commands to the driving board.

DrivingBoardMotorMessage Contains high-level movement instructions:

- distance_to_go → target travel distance
- turning_radius

Purpose: This message defines a movement request from software. The embedded system uses it as input to compute values on the control code.

DrivingBoardMotorPeriodicProgress.proto

Embedded → Software

This message provides runtime feedback during movement execution.
DrivingBoardMotorPeriodicProgress

Contains:

- distance_left → remaining distance to target

Purpose This message allows software to track progress of an ongoing movement command in real time. It is typically sent periodically while a movement is being executed.