Sensor Samples Documentation

---

1. IMU & Navigation Sensors

This section covers samples related to Inertial Measurement Units (IMUs), Gyroscopes, and Odometry sensors used for robot navigation and orientation.

`SensorIMUOrthogonal.java` & `SensorIMUNonOrthogonal.java`

Description:

These samples demonstrate how to use the universal `IMU` interface, which supports both the BNO055 (older Hubs) and BHI260AP (newer Control Hubs) IMUs. They are the modern standard for IMU usage in FTC.

- **Orthogonal:** Assumes the Hub is mounted flat or at 90-degree increments.
- **Non-Orthogonal:** Allows for arbitrary mounting angles.

Key Features:

- Universal `IMU` interface.
- Simplified orientation definition using `RevHubOrientationOnRobot`.
- Retrieval of Yaw, Pitch, and Roll angles.
- Angular velocity measurement.

Code Breakdown:

1. Initialization (Orthogonal):

You must define the `LogoFacingDirection` and `UsbFacingDirection` to match your robot's physical mounting.

```
// Define Hub orientation.

// Default: Logo UP, USB FORWARD

RevHubOrientationOnRobot.LogoFacingDirection logoDirection =
RevHubOrientationOnRobot.LogoFacingDirection.UP;

RevHubOrientationOnRobot.UsbFacingDirection  usbDirection  =
RevHubOrientationOnRobot.UsbFacingDirection.FORWARD;
```

```java
RevHubOrientationOnRobot orientationOnRobot = new
RevHubOrientationOnRobot(logoDirection, usbDirection);



// Initialize the IMU with this mounting orientation

imu.initialize(new IMU.Parameters(orientationOnRobot));
```

2. Initialization (Non-Orthogonal):

For custom mounting, you define rotations (X, Y, Z) to transform the default orientation to your actual mount.

```java
// Define the desired axis rotations.

double xRotation = 0;  // enter the desired X rotation angle here.

double yRotation = 0;  // enter the desired Y rotation angle here.

double zRotation = 0;  // enter the desired Z rotation angle here.



Orientation hubRotation = xyzOrientation(xRotation, yRotation, zRotation);



// Initialize the IMU

RevHubOrientationOnRobot orientationOnRobot = new
RevHubOrientationOnRobot(hubRotation);

imu.initialize(new IMU.Parameters(orientationOnRobot));
```

3. Reading Data:

Retrieve orientation (Yaw/Pitch/Roll) and angular velocity.

```java
// Retrieve Rotational Angles and Velocities

YawPitchRollAngles orientation = imu.getRobotYawPitchRollAngles();
```

```java
AngularVelocity angularVelocity =
imu.getRobotAngularVelocity(AngleUnit.DEGREES);



// Access specific axes

double yaw = orientation.getYaw(AngleUnit.DEGREES);

double pitch = orientation.getPitch(AngleUnit.DEGREES);

double roll = orientation.getRoll(AngleUnit.DEGREES);
```

4. Resetting Yaw:

Crucial for field-centric drive or resetting heading.

```java
if (gamepad1.y) {

    imu.resetYaw();

}
```

`SensorAndyMarkIMUOrthogonal.java` & `SensorAndyMarkIMUNonOrthogonal.java`

Description:

Specific samples for the AndyMark IMU, similar to the universal IMU samples but tailored for AndyMark's hardware wrapper.

Key Features:

- Uses `AndyMarkIMUOrientationOnRobot`.
- Similar logic to the universal IMU samples.

Code Breakdown:

1. Initialization:

Uses `I2cPortFacingDirection` instead of `UsbFacingDirection`.

```java
LogoFacingDirection logoDirection = LogoFacingDirection.UP;

I2cPortFacingDirection i2cDirection = I2cPortFacingDirection.FORWARD;



AndyMarkIMUOrientationOnRobot orientationOnRobot = new
AndyMarkIMUOrientationOnRobot(logoDirection, i2cDirection);

imu.initialize(new IMU.Parameters(orientationOnRobot));
```

`SensorGoBildaPinpoint.java`

Description:

Demonstrates how to use the goBILDA Pinpoint Odometry Computer. This device handles odometry calculations (X, Y, Heading) internally, offloading work from the Control Hub.

Key Features:

- Dedicated hardware for odometry.
- Supports both goBILDA pods and custom encoders.
- Persistent tracking.

Code Breakdown:

1. Initialization & Configuration:

You must set the offsets of your odometry pods relative to the robot's center.

```java
pinpoint = hardwareMap.get(GoBildaPinpointDriver.class, "pinpoint");



// Set offsets in MM (X = forward/back, Y = left/right)

pinpoint.setOffsets(-84.0, -168.0, DistanceUnit.MM);



// Set encoder resolution (e.g., goBILDA 4-Bar Pods)

pinpoint.setEncoderResolution(GoBildaPinpointDriver.GoBildaOdometryPods.goBI
```

```
LDA_4_BAR_POD);


// Set encoder directions

pinpoint.setEncoderDirections(GoBildaPinpointDriver.EncoderDirection.FORWARD
,


GoBildaPinpointDriver.EncoderDirection.FORWARD);



// Reset Position and Calibrate IMU (Do this when stationary!)

pinpoint.resetPosAndIMU();
```

2. Setting Initial Position:

Useful for starting autonomous at a known location.

```
pinpoint.setPosition(new Pose2D(DistanceUnit.INCH, 0, 0, AngleUnit.DEGREES,
0));
```

3. Reading Data:

The `update()` method must be called every loop.

```
pinpoint.update();

Pose2D pose2D = pinpoint.getPosition();



double x = pose2D.getX(DistanceUnit.INCH);

double y = pose2D.getY(DistanceUnit.INCH);

double heading = pose2D.getHeading(AngleUnit.DEGREES);
```

`SensorSparkFunOTOS.java`

Description:

Shows how to use the SparkFun Qwiic Optical Tracking Odometry Sensor (OTOS). This sensor uses an optical mouse-like sensor and an IMU to track position without external encoder wheels.

Key Features:

- Non-contact odometry.
- Built-in IMU.
- Scalar tuning for accuracy.

Code Breakdown:

1. Initialization & Configuration:

```
myOtos = hardwareMap.get(SparkFunOTOS.class, "sensor_otos");


// Set units

myOtos.setLinearUnit(DistanceUnit.INCH);

myOtos.setAngularUnit(AngleUnit.DEGREES);


// Set offset from robot center (x, y, heading)

SparkFunOTOS.Pose2D offset = new SparkFunOTOS.Pose2D(0, 0, 0);

myOtos.setOffset(offset);


// Set calibration scalars (Tuned experimentally)

myOtos.setLinearScalar(1.0);

myOtos.setAngularScalar(1.0);


// Calibrate IMU (Must be stationary)
```

```
myOtos.calibrateImu();

myOtos.resetTracking();
```

2. Reading Position:

```
SparkFunOTOS.Pose2D pos = myOtos.getPosition();

telemetry.addData("X coordinate", pos.x);

telemetry.addData("Y coordinate", pos.y);

telemetry.addData("Heading angle", pos.h);
```

`SensorBNO055IMU.java` & `SensorBNO055IMUCalibration.java` (Legacy)

Description:

These are legacy samples for the BNO055 IMU found in older Expansion Hubs. **It is recommended to use the `SensorIMUOrthogonal` samples instead.**

Key Features:

- Direct `BNO055IMU` interface.
- Manual calibration process (rarely needed now).

Code Breakdown:

```
BNO055IMU.Parameters parameters = new BNO055IMU.Parameters();

parameters.angleUnit = BNO055IMU.AngleUnit.DEGREES;

imu = hardwareMap.get(BNO055IMU.class, "imu");

imu.initialize(parameters);


// Reading angles
```

```java
Orientation angles = imu.getAngularOrientation(AxesReference.INTRINSIC,
AxesOrder.ZYX, AngleUnit.DEGREES);
```

`SensorKLNavxMicro.java`

Description:

Interface for the Kauai Labs navX Micro sensor.

Key Features:

- Uses `NavxMicroNavigationSensor` and `IntegratingGyroscope`.
- High-precision gyroscope.

Code Breakdown:

```java
navxMicro = hardwareMap.get(NavxMicroNavigationSensor.class, "navx");

gyro = (IntegratingGyroscope)navxMicro;



// Calibration loop

while (navxMicro.isCalibrating())  {

    Thread.sleep(50);

}



// Reading data

AngularVelocity rates = gyro.getAngularVelocity(AngleUnit.DEGREES);

Orientation angles = gyro.getAngularOrientation(AxesReference.INTRINSIC,
AxesOrder.ZYX, AngleUnit.DEGREES);
```

`SensorMRGyro.java`

Description:

Legacy sample for the Modern Robotics Gyro.

Key Features:

- `ModernRoboticsI2cGyro` interface.
- Z-axis integrator reset.

Code Breakdown:

```
modernRoboticsI2cGyro = hardwareMap.get(ModernRoboticsI2cGyro.class,
"gyro");

modernRoboticsI2cGyro.calibrate();



// Reset Z Axis

if (gamepad1.a && gamepad1.b) {

    modernRoboticsI2cGyro.resetZAxisIntegrator();

}



int heading = modernRoboticsI2cGyro.getHeading();
```

2. Vision, Color & Distance Sensors

This section covers sensors that detect light, color, distance, and visual targets (AprilTags).

`SensorColor.java` & `SensorMRColor.java`

Description:

Demonstrates how to use a standard Color Sensor (like REV Color Sensor V3) or the legacy Modern Robotics Color Sensor.

Key Features:

- `NormalizedColorSensor` interface (preferred).
- HSV (Hue, Saturation, Value) conversion.
- Gain adjustment.

- LED control.

Code Breakdown:

1. Initialization:

```
colorSensor = hardwareMap.get(NormalizedColorSensor.class, "sensor_color");


// Enable LED (if supported)

if (colorSensor instanceof SwitchableLight) {

  ((SwitchableLight)colorSensor).enableLight(true);

}
```

2. Reading Data & HSV Conversion:

It's best to convert RGB to HSV for more robust color detection.

```
// Get normalized colors

NormalizedRGBA colors = colorSensor.getNormalizedColors();


// Convert to HSV

final float[] hsvValues = new float[3];

Color.colorToHSV(colors.toColor(), hsvValues);


telemetry.addData("Hue", hsvValues[0]);

telemetry.addData("Saturation", hsvValues[1]);

telemetry.addData("Value", hsvValues[2]);
```

3. Gain Adjustment:

Increasing gain helps in low light but can saturate the sensor.

```
colorSensor.setGain(gain);
```

`SensorHuskyLens.java`

Description:

Interface for the DFRobot HuskyLens, an AI vision sensor capable of detecting tags, objects, and colors.

Key Features:

- `HuskyLens` hardware class.
- Algorithm selection (Tag Recognition, Object Recognition, etc.).
- Block-based detection results.

Code Breakdown:

1. Initialization & Algorithm Selection:

```
huskyLens = hardwareMap.get(HuskyLens.class, "huskylens");


// Select Algorithm (e.g., TAG_RECOGNITION)

huskyLens.selectAlgorithm(HuskyLens.Algorithm.TAG_RECOGNITION);
```

2. Reading Blocks:

The sensor returns an array of "Blocks" representing detected objects.

```
HuskyLens.Block[] blocks = huskyLens.blocks();


for (HuskyLens.Block block : blocks) {

    telemetry.addData("Block", block.toString());
```

```
    // Access properties: block.x, block.y, block.width, block.height,
block.id

}
```

`SensorLimelight3A.java`

Description:

Demonstrates usage of the Limelight 3A vision sensor. It provides high-speed tracking of AprilTags, Neural Networks, and more.

Key Features:

- `Limelight3A` interface.
- Pipeline switching.
- Rich result data (Botpose, Fiducials, Classifiers).

Code Breakdown:

1. Initialization:

```java
limelight = hardwareMap.get(Limelight3A.class, "limelight");

limelight.pipelineSwitch(0); // Switch to pipeline 0

limelight.start(); // Start polling
```

2. Reading Results:

```java
LLResult result = limelight.getLatestResult();

if (result.isValid()) {

    // Get Robot Pose (3D)

    Pose3D botpose = result.getBotpose();

    // Get Fiducials (AprilTags)

    List<LLResultTypes.FiducialResult> fiducials =
```

```java
result.getFiducialResults();

    for (LLResultTypes.FiducialResult fr : fiducials) {

        // fr.getFiducialId(), fr.getTargetXDegrees(), etc.

    }

}
```

`SensorREV2mDistance.java`, `SensorAndyMarkTOF.java`, `SensorMRRangeSensor.java`

Description:

These samples demonstrate various distance sensors.

- **REV 2m & AndyMark TOF:** Time-of-Flight laser sensors (high accuracy).
- **MR Range Sensor:** Ultrasonic + Optical combination.

Key Features:

- `DistanceSensor` interface (common to all).
- `getDistance(DistanceUnit unit)` method.

Code Breakdown:

1. Reading Distance:

The core method is identical for most distance sensors.

```java
sensorDistance = hardwareMap.get(DistanceSensor.class, "sensor_distance");



// Read distance in specific units

double distMM = sensorDistance.getDistance(DistanceUnit.MM);

double distCM = sensorDistance.getDistance(DistanceUnit.CM);

double distInch = sensorDistance.getDistance(DistanceUnit.INCH);
```

2. Sensor Specifics (REV 2m):

```
Rev2mDistanceSensor sensorTimeOfFlight = (Rev2mDistanceSensor)
sensorDistance;

boolean timedOut = sensorTimeOfFlight.didTimeoutOccur();
```

3. Sensor Specifics (MR Range):

```
// Access raw ultrasonic/optical values

telemetry.addData("raw ultrasonic", rangeSensor.rawUltrasonic());

telemetry.addData("raw optical", rangeSensor.rawOptical());
```

`SensorMROpticalDistance.java`

Description:

Legacy Modern Robotics Optical Distance Sensor (ODS). It measures reflected light intensity, not true distance (except at very close range).

Code Breakdown:

```
odsSensor = hardwareMap.get(OpticalDistanceSensor.class, "sensor_ods");

double lightLevel = odsSensor.getLightDetected(); // 0.0 to 1.0
```

3. Touch, LED & Encoders

This section covers Touch Sensors, LED Drivers, and advanced Encoder modules like the OctoQuad.

`SampleRevBlinkinLedDriver.java`

Description:

Demonstrates how to control the REV Blinkin LED Driver, which creates various lighting patterns.

Key Features:

- `RevBlinkinLedDriver` interface.
- Preset patterns (Rainbow, Heartbeat, etc.).
- Manual and Auto modes.

Code Breakdown:

1. Initialization:

```
blinkinLedDriver = hardwareMap.get(RevBlinkinLedDriver.class, "blinkin");

pattern = RevBlinkinLedDriver.BlinkinPattern.RAINBOW_RAINBOW_PALETTE;

blinkinLedDriver.setPattern(pattern);
```

2. Changing Patterns:

```
// Next pattern

pattern = pattern.next();

blinkinLedDriver.setPattern(pattern);



// Previous pattern

pattern = pattern.previous();

blinkinLedDriver.setPattern(pattern);
```

`SensorTouch.java` & `SensorDigitalTouch.java`

Description:

Demonstrates how to use a standard Touch Sensor (REV Touch Sensor, Magnetic Limit Switch) or a generic Digital Channel.

Key Features:

- `TouchSensor` interface (simple `isPressed()`).

- `DigitalChannel` interface (more generic, `getState()`).

Code Breakdown:

1. TouchSensor (Simple):

```
touchSensor = hardwareMap.get(TouchSensor.class, "sensor_touch");


if (touchSensor.isPressed()) {

    telemetry.addData("Touch Sensor", "Is Pressed");

}
```

2. DigitalChannel (Generic):

```
digitalTouch = hardwareMap.get(DigitalChannel.class, "digitalTouch");

digitalTouch.setMode(DigitalChannel.Mode.INPUT);


// getState() returns true if HIGH (not pressed for active low), false if LOW (pressed)

if (digitalTouch.getState() == false) {

    telemetry.addData("Button", "PRESSED");

}
```

`SensorOctoQuad.java` (Basic)

Description:

Shows how to use the DigitalChickenLabs OctoQuad to read multiple encoders (e.g., for odometry pods).

Key Features:

- `OctoQuad` hardware class.
- Reading positions and velocities for up to 8 channels.
- Resetting encoders.

Code Breakdown:

1. Initialization:

```
octoquad = hardwareMap.get(OctoQuad.class, "octoquad");


// Configure directions

octoquad.setSingleEncoderDirection(0, OctoQuad.EncoderDirection.REVERSE);


// Set velocity sample interval (e.g., 50ms)

octoquad.setAllVelocitySampleIntervals(50);


// Reset positions

octoquad.resetAllPositions();
```

2. Reading Data (Caching):

Using caching methods is more efficient for I2C.

```
int posLeft = octoquad.readSinglePosition_Caching(0);

int velLeft = octoquad.readSingleVelocity_Caching(0);
```

`SensorOctoQuadAdv.java` (Advanced)

Description:

Demonstrates advanced OctoQuad features, specifically for a Swerve Drive setup. It handles both Quadrature Encoders (Drive) and Pulse-Width Absolute Encoders (Steer).

Key Features:

- Configuring Channel Banks (Quad vs Pulse Width).
- Reading all encoder data in one block (`readAllEncoderData`).
- Pulse-width parameter configuration.

Code Breakdown:

1. Configuration:

```
// Bank 1: Quadrature, Bank 2: Pulse Width

octoquad.setChannelBankConfig(OctoQuad.ChannelBankConfig.BANK1_QUADRATURE_BANK2_PULSE_WIDTH);



// Configure Pulse Width for REV Through Bore Encoder (1-1024us)

octoquad.setSingleChannelPulseWidthParams(4, new OctoQuad.ChannelPulseWidthParams(1, 1024));
```

2. Efficient Reading:

Reads all 8 channels (position + velocity) in a single I2C transaction.

```
OctoQuad.EncoderDataBlock encoderDataBlock = new OctoQuad.EncoderDataBlock();

octoquad.readAllEncoderData(encoderDataBlock);



// Access data

double driveCounts = encoderDataBlock.positions[0];

double steerPos = encoderDataBlock.positions[4];
```

`SensorOctoQuadLocalization.java` (MK2 Only)

Description:

Demonstrates the "Absolute Localizer" feature of the OctoQuad MK2, which has a built-in IMU and performs odometry calculations on-board.

Key Features:

- On-board odometry (X, Y, Heading).
- IMU calibration.
- Parameter tuning (Ticks per MM, Offsets).

Code Breakdown:

1. Configuration:

You must set physical parameters for accurate tracking.

```
oq.setLocalizerCountsPerMM_X(12.34f);

oq.setLocalizerTcpOffsetMM_X(123.4f);

oq.resetLocalizerAndCalibrateIMU();
```

2. Reading Localizer Data:

```
OctoQuad.LocalizerDataBlock localizer = new OctoQuad.LocalizerDataBlock();

oq.readLocalizerData(localizer);



if (localizer.crcOk) {

    telemetry.addData("X", localizer.posX_mm);

    telemetry.addData("Y", localizer.posY_mm);

    telemetry.addData("Heading", localizer.heading_rad);
```

```
  }
}
```