OctoQuad Localizer Quickstart

Absolute Localizer Introduction

The OctoQuad FTC Edition MK2 features an onboard IMU and absolute localizer algorithm designed for use with encoders on free-spinning position tracking wheels ("deadwheel odometry pods") on an FTC robot. The OctoQuad functions as a coprocessor, combining encoder data and IMU data at 1.92KHz to calculate absolute position. This provides excellent position tracking accuracy which is independent of user code execution speed on the host processor. Absolute position data can be queried by the host processor over the I2C interface, along with a cyclic-redundancy-check (CRC) signature to verify the integrity of the received data.

GitHub Repository

This document refers to several files located in the OctoQuad GitHub repository: https://github.com/DigitalChickenLabs/OctoQuad

Absolute Localizer Necessary Configuration

In order to make use of the absolute localizer, the OctoQuad must be informed of some characteristics of your robot and its deadwheel setup.

- Deadwheel encoder ports
- Encoder port directions
- Deadwheel resolutions
- TCP offset
- IMU Scalar

To obtain these values, refer to the setup outline below.

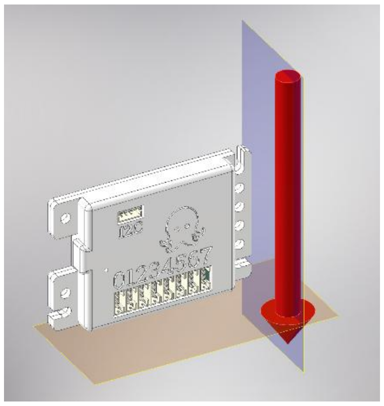
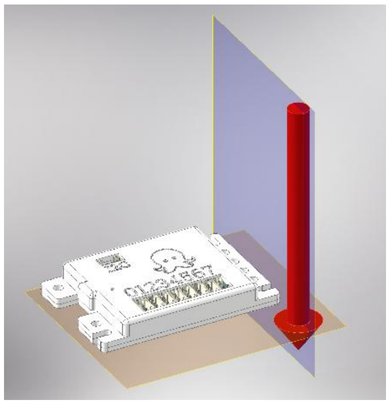
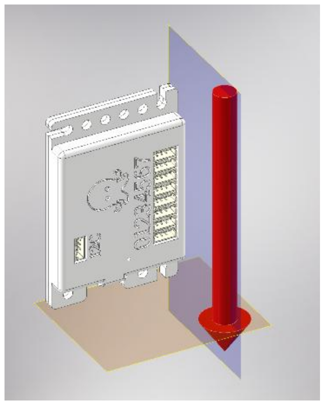
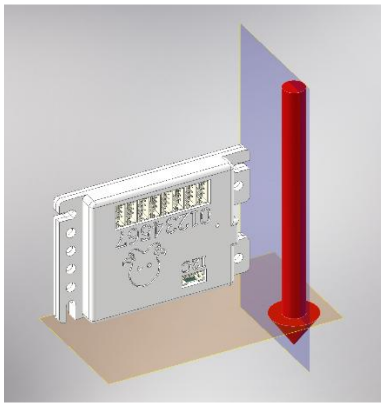
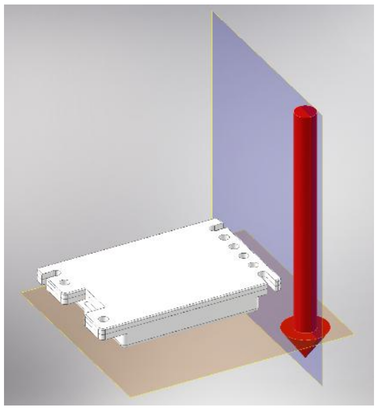
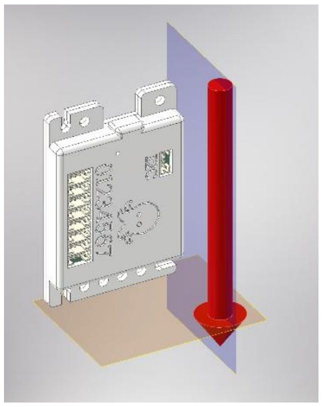
Initial Setup / Calibration Outline

1. Mount the OctoQuad on the robot. Refer to *Mounting the OctoQuad on Your Robot*.
2. Mount the deadwheels on the robot, one for the X axis and one for the Y axis. Refer to *Absolute Coordinate System*.
3. Connect the deadwheel encoders to the OctoQuad, making note of the ports on the OctoQuad that you connect them to.
4. Determine if the direction of one (or both) deadwheel encoders must be reversed. Refer to *Deadwheel Direction Configuration*.

5. Measure the linear resolution of the deadwheel encoders. Refer to *Deadwheel Resolution Configuration*
6. Measure the IMU heading scalar. Refer to *IMU Scalar Calibration*.
7. Measure the TCP offset. Refer to *Localizer TCP Offset*.
8. Plug the determined constants into the LocalizerTest OpMode located in the code_examples/FTC directory of the github repository and the test localizer accuracy.

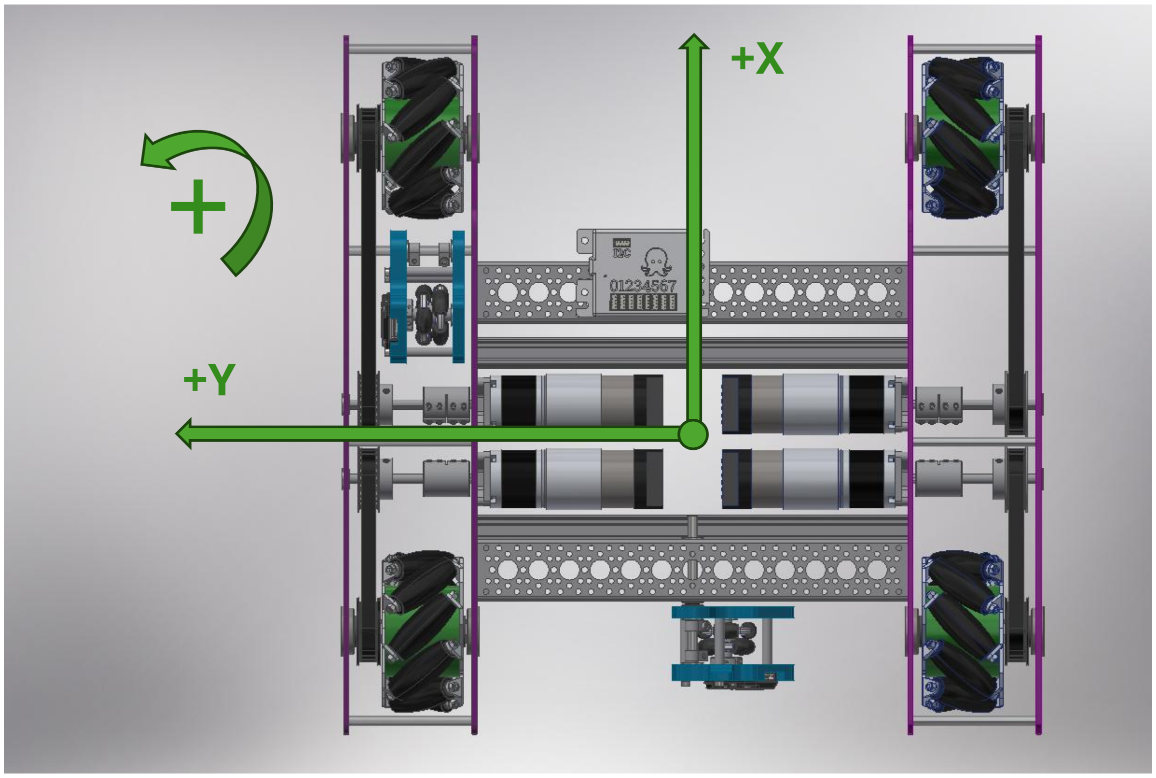
Mounting the OctoQuad on Your Robot

The OctoQuad may be mounted on the robot in any of the 6 orientations orthogonal to the direction of gravity. During IMU calibration, the direction of gravity is automatically detected and the appropriate configuration applied to the internal localizer algorithm. The following graphics illustrate acceptable mounting orientations with respect to gravity:

Absolute Coordinate System

The absolute coordinate system used by the OctoQuad is shown in the figure below. After resetting the localizer, the +X axis points out the front of the robot, and the +Y axis points out the left side of the robot. Counter-clockwise rotation is considered positive. This convention, although at first seemingly counter-intuitive, is chosen for consistency with the cartesian coordinate plane: after calibration, the IMU heading is reset to zero, and moving at an angle of 0 degrees on the cartesian plane means moving in the +X direction.



Deadwheel Port Numbers Configuration

The OctoQuad must be informed of the encoder port numbers to which the deadwheels are connected. To do so, use the following driver function calls when initializing the device:

```
void setLocalizerPortX(int port)
```

```
void setLocalizerPortY(int port)
```

Deadwheel Direction Configuration

The direction of the encoder ports to which the deadwheels are connected must be configured such that, with the robot oriented at 0 degrees, pushing the robot in the +X direction results in an increasing encoder count on the X deadwheel, and likewise, pushing the robot in the +Y direction results in an increasing encoder count on the Y deadwheel (see *Absolute Coordinate System*). A helper OpMode is available to aid in determining this, as well as the deadwheel resolution (see below). If reversing is necessary, use the following driver function call for each deadwheel when initializing the device:

```
void setSingleEncoderDirection(int port, EncoderDirection direction)
```

Deadwheel Resolution Configuration

Because the number of deadwheel encoder ticks generated for a given robot travel distance depends on both the resolution of the encoder and diameter of the wheel, the localizer must be informed (for both axes) of how many encoder ticks correspond to one millimeter of travel. Although a theoretical value may be calculated for this based on the hardware used on the robot, it is NOT recommended to do so. Rather, an empirical measurement over several meters of travel is suggested. An OpMode is available which provides a guided process for completing these steps (DeadwheelCalibator in the code_examples/FTC directory of the GitHub repository). Alternatively, if you already know these constants based on specifications provided by the deadwheel manufacturer, you may use those constantly directly instead. For example, values for GoBilda odometry pods are available here. Once the constants have been determined, they may be applied using the following driver function calls when initializing the device:

```
void setLocalizerCountsPerMM_X(float ticksPerMM_x)
```

```
void setLocalizerCountsPerMM_Y(float ticksPerMM_y)
```

IMU Scalar Calibration

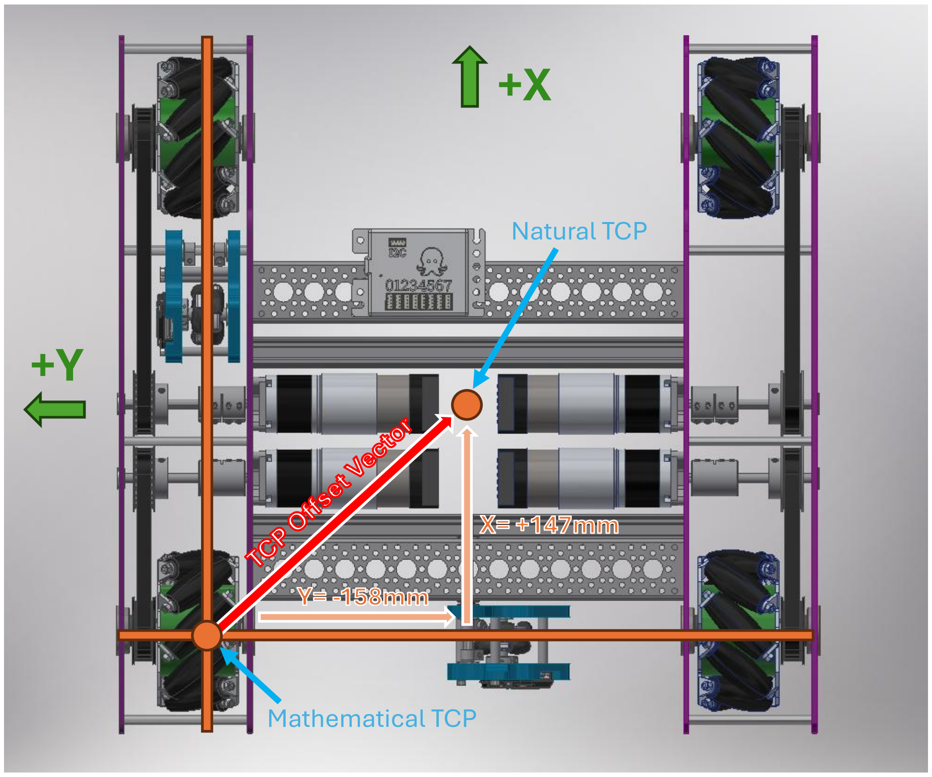
The OctoQuad self-calibrates the zero-angular-velocity bias of the IMU. However, it cannot automatically calibrate the angular velocity scale factor for the IMU. It is critical to calculate this scale factor in order to obtain accurate heading data. Incorrect scalars will not cause heading drift – but they will result in the IMU reporting more/less degrees of rotation than it was truly rotated. An OpMode is available which provides a guided process for completing this calibration (HeadingScalarCalibrator in the code_examples/FTC directory of the GitHub repository). Once the scale factor has been determined, it may be set using the following driver function call:

```
void setLocalizerImuHeadingScalar(float headingScalar)
```

Localizer TCP Offset

The Tracking Center Point (TCP) is the “datum point” on the robot to which the robot’s XY displacement relative to the origin of the absolute coordinate system is measured. An important characteristic of the TCP is that it is the point about which a rotation does not influence the reported displacement. Imagine putting an orange sticker on a corner of the robot (call this the TCP). If you then rotate the robot with the orange sticker being the center of rotation, the physical location of the orange sticker will not change. If you measure the displacement of the orange sticker relative to the playing field before and after rotating the robot, it will be exactly the same.

With deadwheel odometry, the initial mathematical location of the TCP is at the intersection of lines drawn through the deadwheels and parallel to their respective direction of travel:



The initial mathematical TCP is located at this intersection because it is the location where, if it is the center of rotation, the odometry wheels experience no normal motion (only tangential). Said another way, rotating about this point causes the odometry wheels to exclusively slide laterally rather than rotate axially. Thus, because there is no axial rotation, the reported XY displacement remains constant.

This initial mathematical location of the TCP can prove rather inconvenient, because it is NOT coincident with the robot's natural center of rotation, i.e. its geometric center. This means that, initially, the displacement of the robot in the absolute coordinate system is not being measured relative to its natural center of rotation. This has the effect of the reported X and Y positions "wobbling" in a sinusoidal fashion if the robot is rotated about its natural center of rotation. Thus, it is often convenient to, in software, relocate the TCP to be coincident with the robot's geometric center.

The TCP offsets are defined as the vector which points from the initial mathematical TCP location to the desired/natural TCP location, in the coordinate system defined in *Absolute Coordinate System*. In the example situation shown in the figure, the X offset is +147mm and the Y offset is -158mm because to relocate the mathematical TCP to the natural TCP, it must be moved <+147, -158> in the coordinate system. These distances can be measured with a tape measure; it is not critical to tracking accuracy for them to be exact.

Once the desired offsets have been determined, they may be applied using the following driver function calls when initializing the device:

```
void setLocalizerTcpOffsetMM_X(float tcpOffsetMM_X)
```

```
void setLocalizerTcpOffsetMM_Y(float tcpOffsetMM_Y)
```

Localizer Velocity Calculation Interval

The localizer calculates the velocity in the absolute coordinate frame at a fixed interval (by default, 25ms). Longer periods give higher resolution with more latency, shorter periods give lower resolution with less latency. If you wish to change this interval from its default value, use the following driver function call when initializing the device:

```
void setLocalizerVelocityIntervalMS(int ms)
```

CRC Data Integrity Check

The I2C bus protocol does not have any data integrity guarantee mechanisms built into it by default. In FTC, ESD events can cause corrupted I2C data. Additionally, it has been observed that the Control Hub and Expansion Hub appear to have a hardware bug wherein motor switching noise can cause the I2C peripheral to become confused and send invalid I2C clock pulses that violate timing, which also causes corrupted data. One frequently observed symptom of this is getting "NaN" results from an I2C device. In order to mitigate these issues, the OctoQuad supplies a CRC signature along with the data in order to validate the integrity of the data on the receiving end.

Both the `EncoderDataBlock` and `LocalizerDataBlock` data structures contain a `crcOk` boolean field which reports whether or not the data is valid. If this flag is not set, you should throw away the data and attempt a read again.

Device Initialization

When initializing the device from user code, follow the following procedure (this is also illustrated in the LocalizerTest OpMode located in the code_examples/FTC directory of the github repository):

1. Retrieve the device handle from the HardwareMap
2. Configure all necessary localizer parameters (refer to *Absolute Localizer Necessary Configuration*)
3. Apply the parameters and calibrate the IMU by calling `resetLocalizerAndCalibrateIMU()`
4. Wait for the localizer status to report RUNNING before querying position data
5. Make sure to check `crcOk` when polling position data!