The Pinpoint Odometry Computer is a coprocessor meant for competition robotics applications. It reads two “dead wheel” odometry pods (encoders which read the position of a non-powered wheel) and an internal IMU to track the position and orientation of a robot in near-real time.

It outputs processed information through an I²C communication interface.

Summary of Product Ratings				
Input Voltage	3.3V - 5V		Max IMU Rotation Speed	2000RPM
Input Current	80mA		Algorithm Update Frequency	1500hz
Maximum Encoder Speed	256,000 events/second		Connector Type	4-Pos JST PH [FH-MC]
I²C Address	0x31		I²C Max Frequency (tested)	400khz

Table of Contents:

Pinpoint Required Configuration: 2
Pinpoint Optional Configuration: 2-3
Using the Pinpoint in FTC: 4
Zeroing/Calibrating the IMU: 4
Device State Codes: 4
Step by Step Setup Guide: 5
Device Outputs: 6
Using setPosition: 6
Troubleshooting: 7
FTC Programming: Functions 8-11
FTC Programming: Pose2D 12
FTC Programming: Enums 12
I²C Register Map: 13-14

Pinpoint Required Configuration:

This section covers the calibration inputs for the Pinpoint. These are all important to ensure accurate tracking. Device Configurations are lost on device power cycles, so your I²C Controller device should send the config information over I²C once every time the code starts.

1. **Encoder Directions**
   The most important step in configuring your Pinpoint is ensuring encoder direction is correct. The X encoder/odometry pod should increase in count when the robot is moved forward, and the Y encoder/pod should increase when the robot is moved to the left. If either of these are backwards, reverse them.

2. **Encoder Resolution**
   The Pinpoint works with a wide variety of odometry pods; to allow this, you need to configure the resolution of the particular pod you are providing. This resolution is expected in "ticks per mm". This depends on the diameter of the tracking wheel and the quadrature resolution of the encoder you're using. To find this number divide the CPR (counts per rotation) of your encoder by the circumference of the tracking wheel (in mm). For example, the goBILDA® 4-Bar Odometry Pod (SKU: 3110-0001-0002) uses a 2000 CPR encoder with a 32mm diameter wheel, so you'd divide 2000/100.53 to find the ticks per mm of 19.894.

Pinpoint Optional Configuration:

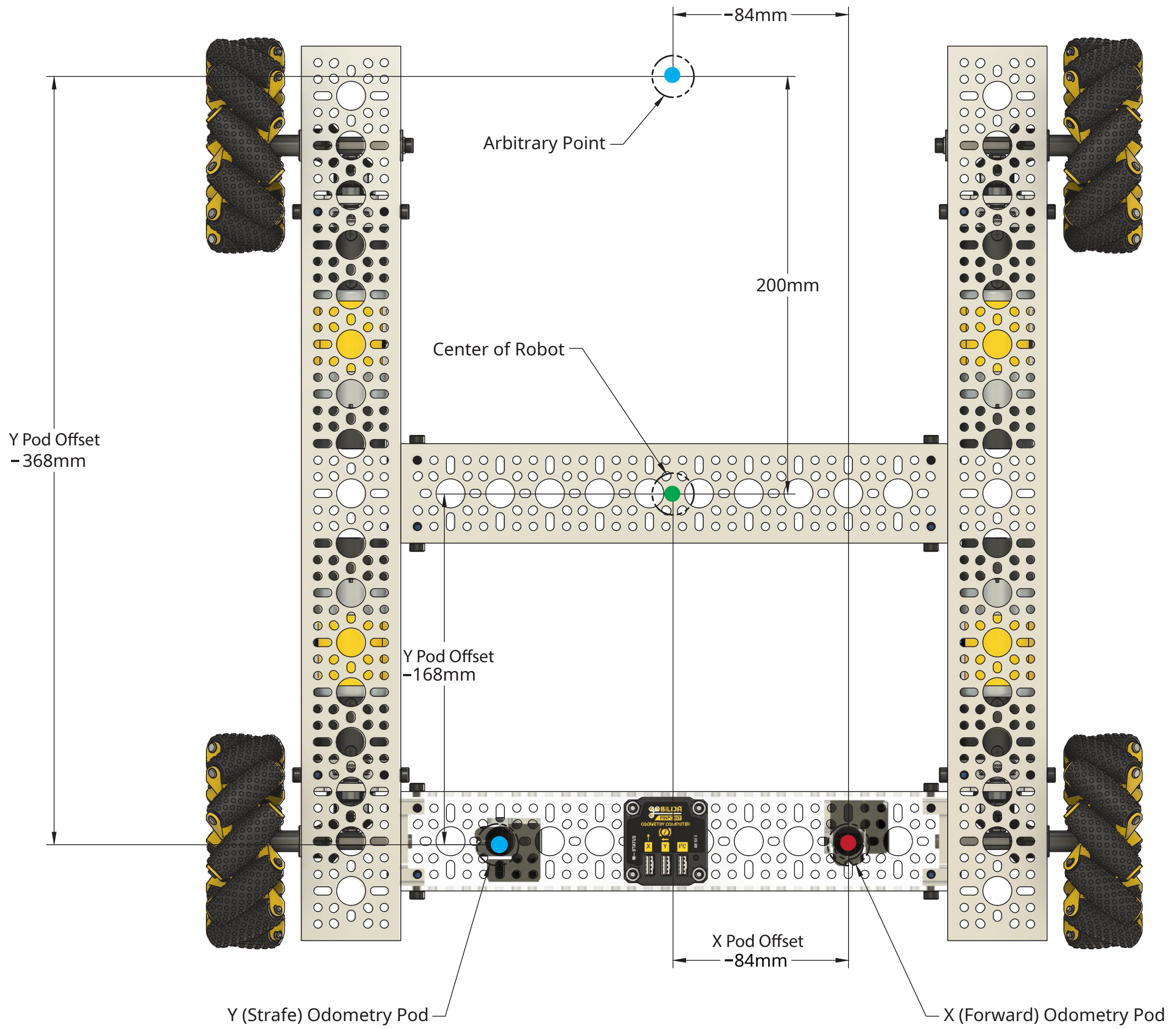
1. **Pod Offsets (see next page for graphic)**
   The Pinpoint works best when you also tell it the position of each odometry pod on your robot. You can send an "X Pod Offset" and a "Y Pod Offset".
   These two variables describe where the two odometry pods are located on your robot, relative to a "tracking point". This tracking point is the single point that the Pinpoint reports the location of. Many users prefer this to be the center of the robot, but critically with a two dead-wheel odometry setup, the position of this tracking point on the robot does not affect the accuracy of the system. So while it is often put it in the center of their robot, it can be at any arbitrary point.

   These offsets are the distance from the center of the omni wheel, perpendicular to wheel's travel, to the tracking point. The distance along the length of the wheel's primary axis of travel are not important. This means that the X Pod Offset actually measures how far in the Y axis that the X pod is mounted.

   The X Pod Offset measures how far (in mm) sideways the X pod is from the tracking point. Left of the point is a positive number, right of the point is a negative number.
   The Y Pod Offset measures how far forwards the Y pod is from the tracking point. Forwards of the point is positive, backwards is negative. In this example, to track the center of your robot, the X offset should be -84mm, and the Y offset should be -168mm.

   Some users may choose to put this tracking point at a part of their robot more reflective of the end use. In this example we've moved it 200mm forward of the center of the robot, to reflect something like an intake. In that case, our X offset should still be -84mm, but the Y offset should be -368mm.

2. **Yaw Scalar (Factory Calibrated, Tuning discouraged)**
   This is a scalar which is applied to the heading of the device. This is meant to compensate for small device-to-device differences in exactly how much rotation the IMU reports. This value is calibrated on each device before it is shipped to the user, so it should be very accurate. If you set the Yaw Scalar to 1.1, it will record 1.1 degree of movement when the internal sensor reads 1 degree. The factory calibration for each Pinpoint is written to device memory. On a power cycle, it will revert to factory calibration until you send it a new Yaw Scalar to use.


-84mm
Arbitrary Point
200mm
Center of Robot
Y Pod Offset
-368mm
Y Pod Offset
-168mm
X Pod Offset
-84mm
Y (Strafe) Odometry Pod
X (Forward) Odometry Pod

Using the Pinpoint in FTC:

We provide a Java driver for FTC that allows you to get up-and-running very quickly with this device without having to implement the I2C protocol. Refer to the GitHub driver for more detailed information.
https://github.com/goBILDA-Official/FtcRobotController-Add-Pinpoint/

Zeroing/Calibrating the IMU:

The Pinpoint internal IMU calibrates its "zero offset". This allows the Pinpoint to accurately track the robot heading. It is critical that the robot be stationary when this zero offset measurement is taken. A "bad" zero offset will result in the estimated heading constantly drifting while the robot is stationary. This calibration takes aproximately 0.25 seconds.
The Pinpoint captures a set of readings to create a zero offset first when the device first receives power, but it can be manually triggered through the I²C Interface.

FTC: Call recalibrateIMU() to run this calibration.
OR call resetPosAndIMU() to set the estimated position to 0, and recalibrate the IMU. This is recommended when you start the first OpMode of your match (normally Autonomous) as a bad initial zero offset can cause a drift which will move your robot's estimated position, without robot movement happening. Once you have called resetPosAndIMU() when you know your robot is stationary, you can call recalibrateIMU() without overwriting the current estimated position.

Device State Codes:

The Pinpoint has several device "states" that it reports to the user through the I²C interface, and through the RGB LED on the physical device.

Status LED	Device Status Name & Description
Green	Ready - Device is working normally
Red	Calibrating - The device is calibrating it's zero offset and outputs are on hold
Red	Not Ready - The device is first starting up starting from a power cycle
Purple	No Pods Detected - The device does not detect any encoders plugged in
Blue	X Pod Not Detected - The device does not detect an X pod plugged in
Orange	Y Pod Not Detected - The device does not detect a Y pod plugged in

Step by Step Setup Guide:

This will walk you through setting up an Odometry Computer on a robot with two Odometry Pods.

1. Mount your Pinpoint sticker-side up on the chassis.
2. Mount two odometry pods on the robot, with one tracking forwards (the X pod) and one tracking sideways (the Y pod).
3. Power up your system and ensure you're reading data from the device. We'll be referencing the estimated position of the robot in these steps.
4. Calculate and set the Encoder Resolution calibration for the odometry pods you're using.
   If you're in FTC, and using goBILDA® pods, just tell it which one. That math is already done.
5. Make sure that the heading and position readings of the robot are stable. They should not move while the robot isn't moving. If they do, send a resetPosAndIMU command, or click the Reset button on the Pinpoint.
6. Check that the Device Status is Ready or that the LED on the Pinpoint is green.
   If it is Purple, Blue, or Orange, check your pods and make sure they're correctly connected.
7. Move the robot forward without rotating it. Make sure that the estimated X position **increases.**
   If X decreases, reverse the X Encoder direction by setting the X Encoder Direction to reversed.
   If Y moves more than X, make sure that the pod plugged into the X port is tracking forward.
   If neither value moves, make sure that your X pod is working.
8. Move the robot left without rotating it. Make sure that the Y position increases.
   If Y decreases, reverse the Y Encoder Direction.
   If neither value moves, make sure that your Y pod is working.
9. Measure your Pod Offsets as described on page 2/3, and write those offsets to the Pinpoint.
10. Rotate your robot around the tracking point without sliding the robot forward or sideways. The X and Y positions should stay fairly low. If they fluctuate by more than ~100mm or 4", double check your Pod Offsets. The most common issue here is that one of the offsets is positive when it needs to be negative.
11. Rotate the robot a full turn counterclockwise. The heading should read very close to exactly one rotation positive.
12. Using a tape measure, move your robot along the field and measure how far it has moved, and compare it to how far the Pinpoint reports it moving. If these are significantly different, double check your ticks-per-mm configuration.
13. Reset the estimated position (with the button or with your code) and drive the robot around a little bit before returning to the starting point. The values should be close to zero. If everything is working, expect to see your robot return to within about 10mm (~0.5") of the starting position. More is a good indicator that something is wrong.
14. Under normal circumstances, tuning Yaw Offset is not required.

Device Outputs:

The following values are able to be read from the I²C interface of the device:

- **Device ID -** A static value (2) that can be used to ensure that the device is connected.
- **Device Version** - Tracks which version of the Pinpoint you are working with.
- **Device Status** - Reports the current state of the device including any errors.
- **Loop Time** - The length of time that the previous device cycle took.
- **"Raw" Encoder Values** - The unprocessed encoder value plugged into each port.
- **Estimated Robot Position** - The X, Y, and heading position that the Pinpoint estimates your robot is at,
- relative to where it started.
- **Estimated Robot Velocity** - The X, Y, and heading velocity that the Pinpoint is measuring.
- **Configuration Variables Set to the Device** - Such as X/Y Pod Offsets, and the Yaw-Scalar.

Using setPosition:

SetPositon allows a ton of flexibility with how you use the Pinpoint! Here are some examples:

1. Using FTC Field Coordinates

FTC Field Coordinates are the standard way to refer to where a robot, or game element, is on the field. They start at the center of the field at 0,0,0. X increases as you move towards the audience, and Y increases as you move towards the blue alliance.

By default the Pinpoint will track how far your robot has moved since you started the program, or called resetPosAndIMU(). But setPosition allows you to track your robot in field coordinates instead. When your OpMode starts, send a setPosition command to the Pinpoint with the field coordinates of where your robot is right now. So if you are 600mm away from the center of the field in the X axis, 1200mm away from the center in the Y axis, and oriented 90° away from the audience, send those coordinates to the Pinpoint. Now the Pinpoint will report how far away your robot is from the center of the field.

2. Re-localization

The Pinpoint is a relative tracking device. It tracks how far it estimates that your robot has moved since it started. Sometimes small errors can stack up, and sometimes referencing an absolute localization system (like an AprilTag on an FTC Field) can get you back on track. If you have a camera that can read your robot's position with an AprilTag, every once in a while stop and reference where your robot is relative to the Apriltag, and send a setPosition command with that updated position!

Troubleshooting:

If you are having an issue with your Pinpoint Odometry Computer, please refer to the below troubleshooting suggestions to try to solve your issue. If the issue you're having is not documented below, or you are unable to solve it, please don't hesitate to reach out to our tech support!

Status Lights Never Green:
The Pinpoint status light should turn green on normal operation.
If it is always **Purple**, **Blue**, or **Orange**, then the Pinpoint is not detecting one, or both of your odometry pods.This issue is most likely to be a bad wire, but it could be a bad Pinpoint, or a bad odometry pod. If only one pod is detected, switch the X and Y pods. If the error/color changes (from blue to orange, or orange to blue) then the issue is a pod or a wire. Try to swap the wire between the two pods. If the color changes again, then replace the wire. If the issue follows the pod, then you may need to replace the pod.
If the color is always Purple, Blue, or Orange regardless of pods plugged in, reach out to tech support.
If the status light is always **Red**, make sure that the button is not stuck down. If it turns red when you run your program on the controller, then make sure you aren't calling "resetPosAndIMU" or "recalibrateIMU" constantly. If neither of these fix your issue, reach out to tech support.

I^{2}C Data Always 0
FTC: Make sure that your code calls update(); every time you need new data!
Non-FTC: Reach out to tech support

Drifting IMU
A very very slight heading drift can be normal, but the Pinpoint should not constantly drift. It can detect even very very small changes in the environment, and it's normal to see the estimated heading fluctuate even if someone is just walking near the robot.
If you see this issue, click the reset button while you're sure that the robot is not moving. If you still see a heading drift of more than 1° of drift in 1min with the robot stationary, reach out to tech support.

Tracking Inaccuracies
Small tracking inaccuracies are expected in any odometry system. If you consistently see your robot's position correctly reported to within 6mm (~0.25"), and that inaccuracy is consistent, then while additional tuning may be possible, you are likely operating in the normal range of accuracy for dead wheel systems.
If you see large tracking inaccuracies, then it's likely that there is a configuration issue with your setup. Follow steps 6-13 of the setup guide on the previous page. If you still see large tracking inaccuracies, first see if they are consistent. Run the same pre-programmed path over and over. If you see large changes run-to-run, then make sure your IMU isn't drifting, and double check that your pods are solidly plugged in, and are mounted on the robot in a way that they don't wiggle. A loose connection or a poorly mounted pod is the most likely issue behind inconsistent inaccuracies.
If the configuration is set up properly, and you see large, consistent inaccuracies, reach out to tech support for more in-depth troubleshooting.

FTC Programming: Functions

This section details the user-available **functions** for the Pinpoint Driver.

public void **setOffsets**(double **xOffset**, double **yOffset)**

Sets the odometry pod positions relative to the point that the odometry computer tracks around.
The most common tracking position is the center of the robot.
The X pod offset refers to how far sideways (in mm) from the tracking point the X (forward) odometry pod is. Left of the center is a positive number, right of center is a negative number.
the Y pod offset refers to how far forwards (in mm) from the tracking point the Y (strafe) odometry pod is. forward of center is a positive number, backwards is a negative number.
xOffset - how sideways from the center of the robot is the X (forward) pod? Left increases
yOffset - how far forward from the center of the robot is the Y (Strafe) pod? forward increases

> ***Note: overriding functions***
> *You'll notice setEncoderResolution on this list twice, this is because it's the same command, and you can supply two different data types. Either a double that contains the number of ticks per mm. Or select one of a list of options of goBILDA odometry pods.*

public void **setEncoderResolution**(double **ticks_per_mm**)

Sets the encoder resolution in ticks per mm of the odometry pods. You can find this number by dividing the counts-per-revolution of your encoder by the circumference of the wheel.
should be somewhere between 10 ticks/mm and 100 ticks/mm. a goBILDA Swingarm pod is ~13.26291192
ticks_per_mm How many ticks are recorded per mm linearly traveled by the pod

public void **setEncoderResolution** (GoBildaOdometryPods **pods**)

If you're using goBILDA odometry pods, the ticks-per-mm values are stored here for easy access
pods - goBILDA_SWINGARM_POD or goBILDA_4_BAR_POD

public void **setEncoderDirections**(EncoderDirection **xEncoder**, EncoderDirection **yEncoder**)
Reverse the direction of each encoder.
xEncoder - FORWARD or REVERSED, X (forward) pod should increase when the robot is moving forward
yEncoder - FORWARD or REVERSED, Y (strafe) pod should increase when the robot is moving left

public void **recalibrateIMU**()
Recalibrates the Odometry Computer's internal IMU.
Robot **MUST** be stationary
Device takes a large number of samples, and uses those as the gyroscope zero-offset. This takes approximately 0.25 seconds.

FTC Programming: Functions - Continued

public void **resetPosAndIMU**()
Resets the current position to 0,0,0 and recalibrates the Odometry Computer's internal IMU.
Robot **MUST** be stationary.
Device takes a large number of samples, and uses those as the gyroscope zero-offset. This takes approximately 0.25 seconds.

public void **update**()
Call this once per loop to read new data from the Odometry Computer. Data will only update once this is called.

public void **update**(readData **data**)
Call this once per loop to read new data from the Odometry Computer. This is an override of the update() function which allows a narrower range of data to be read from the device for faster read times. This will **only update the robot heading.** No other data is pulled with this function.
data - GoBildaPinpointDriver. readData. ONLY_UPDATE_HEADING

public deviceStatus **getDeviceStatus**()
Device Status stores any faults the Odometry Computer may be experiencing.
Returns: One of the following states:
NOT_READY - The device is currently powering up. And has not initialized yet. RED LED
READY - The device is currently functioning as normal. GREEN LED
CALIBRATING - The device is currently recalibrating the gyro. RED LED
FAULT_NO_PODS_DETECTED - the device does not detect any pods plugged in. PURPLE LED
FAULT_X_POD_NOT_DETECTED - The device does not detect an X pod plugged in. BLUE LED
FAULT_Y_POD_NOT_DETECTED - The device does not detect a Y pod plugged in. ORANGE LED

public Pose2D **getPosition**()
Note: Pose2D Heading is normalized
Returns: a Pose2D containing the estimated position of the robot.

Note: Normalized and Unnormalized Heading
In FTC there are two ways to look at your robot's heading, normalized and unnormalized.
Normalized: Reports a heading of anywhere between -180° and 180°. If your robot starts at 0°, and then rotates clockwise exactly one half rotation, then it will show 180°. If you rotate two more degrees clockwise, you'll see a normalized heading "wrap" back around to -178°. Reflecting that your heading is now described as 178° counterclockwise from the start.
Unnormalized: Heading will continuously count in the direction you're moving. If you rotate 182° clockwise, this will read 182° If you rotate 10 full turns, your robot will report 3600°. Contrasted to a normalized heading, which would report 0° for a full rotation. No matter how many times your robot has rotated. Normalized and unnormalized headings are both useful! Just in different applications.

FTC Programming: Functions - Continued

public Pose2D **getVelocity**()
Returns: a Pose2D containing the estimated velocity of the robot, velocity is unit per second.

public int **getLoopTime**()
Checks the Odometry Computer's most recent loop time. If values less than 500, or more than 1100 are commonly seen here, there may be something wrong with your device. Please reach out to tech@gobilda.com
Returns: loop time in microseconds (1/ 1,000,000 seconds)

public double **getFrequency**()
Checks the Odometry Computer's most recent loop frequency.
If values less than 900, or more than 2000 are commonly seen here, there may be something wrong with your device. Please reach out to tech@gobilda.com
Returns: Pinpoint Frequency in Hz (loops per second)

public double **getPosX**()
This is used to construct the Pose2D returned in getPosition(). But you can also call it separately.
Returns: Estimated X Position of the robot in mm.

public double **getPosY()**
This is used to construct the Pose2D returned in getPosition(). But you can also call it separately.
Returns: Estimated Y Position of the robot in mm.

public double **getHeading()**
This is used to construct the Pose2D returned in getPosition(). But you can also call it separately.
Returns: An unnormalized estimate of the heading of the robot in radians.

public double **getVelX**()
This is used to construct the Pose2D returned in getVelocity(). But you can also call it separately.
Returns: Estimated X Velocity of the robot in mm/sec.

public double **getVelY**()
This is used to construct the Pose2D returned in getVelocity(). But you can also call it separately.
Returns: Estimated Y Velocity of the robot in mm/sec.

public double **getVelH**()
This is used to construct the Pose2D returned in getVelocity(). But you can also call it separately.
Returns: Estimated Heading Velocity of the robot in radians/sec.

public int **getEncoderX**()
This is an unmodified report of the current count of the X encoder. This is not reversed with setEncoderDirection(), or changed by setEncoderResolution.
Returns: Raw X Encoder Value.

FTC Programming: Functions - Continued

public int **getEncoderY**()
This is an unmodified report of the current count of the Y encoder. This is not reversed with setEncoderDirection(), or changed by setEncoderResolution.
Returns: Raw Y Encoder Value.

Note: Returns not read in update()
Most of the commonly used variables that the Pinpoint outputs are read at once with the update() function, but some aren't. These are marked with (Uses its own I²C Read). ***Avoid calling these functions every cycle****. They add about 2ms each to your loop time. They are great to read once when you are waiting for the user to click Start (to check that your configurations have correctly been applied), but avoid calling them inside your main loop.*

public float **getXOffset**()
Uses its own I²C Read.
Returns: The X Pod Offset that is currently being used by the device.

public int **getDeviceID**()
Uses its own I²C Read.
Checks the deviceID of the Pinpoint. Should return 2 if functional.
Note: because this is not read using update(), avoid calling this every loop. As it requires it's own I²C read. Which adds time to your cycle.
Returns: goBILDA Internal deviceID.

public int **getDeviceVersion**()
Uses its own I²C Read.
Note: because this is not read using update(), avoid calling this every loop. As it requires it's own I²C read. Which adds time to your cycle.
Returns: The version of the Pinpoint you're using

public float **getYawScalar**()
Uses its own I²C Read.
Returns: The yaw scalar currently being used by the device. This is pre-calibrated. If this number is less than 0.9, or more than 1.1. And you have not overwritten the default with your code. Reach out to tech@gobilda.com.

FTC Programming: Pose2D

Pose2D is the primary way that you get position data from the Pinpoint. It represents the position and heading of an object in 2d space as two distances and a rotation. It's especially nice because it lets you get the X/Y and heading in about whatever unit you prefer to work in. The Pinpoint outputs mm and radians. Once you are using Pose2D, however, you can call getX() in inches, or even meters.

Pose2D(DistanceUnit **distanceUnit**, double **x**, double **y**, AngleUnit **headingUnit**, double **heading**)

When you create a Pose2D, you need to specify a distance unit and an angle unit, and provide two distances and a heading in those units.
A distance unit is one of the following:
METER, CM, MM, INCH.
And an angle unit is one of the following:
DEGREES, RADIANS.

Once you have a Pose2D, you can pull the data out of it by calling one of three functions:

double **getX**(DistanceUnit **unit**)
Returns: The X distance of the Pose2D in the AngleUnit specified.

double **getY**(DistanceUnit **unit**)
Returns: The Y distance of the Pose2D in the AngleUnit specified.

double **getHeading**(AngleUnit **unit**)
Returns: The angle of the Pose2D in the AngleUnit specified.

FTC Programming: Enums

The Pinpoint creates a few public Enums.

public enum **DeviceStatus**{
NOT_READY
READY
CALIBRATING
FAULT_X_POD_NOT_DETECTED
FAULT_Y_POD_NOT_DETECTED
FAULT_NO_PODS_DETECTED}

public enum **EncoderDirection**{
FORWARD
REVERSED}

public enum **GoBildaOdometryPods**{
goBILDA_SWINGARM_POD
goBILDA_4_BAR_POD}

I^{2}C Register Map:

If you're writing a custom driver for the Pinpoint, refer to the register map below. It outlines the information contained in each register address.

Register Address	Register Name	Data Type	Length	Readability	Notes
1	Device ID	UInt32	4 bytes	Read	Should be 2
2	Device Version	UInt32	4 bytes	Read	Starts at 1
3	Device Status	UInt32	4 Bytes	Read	See Below
4	Device Control	UInt32	4 Bytes	Write	See Below
5	Loop Time	UInt32	4 Bytes	Read	Time in µsec of last device loop
6	X Encoder Value	UInt32	4 Bytes	Read	Raw Value of X Encoder
7	Y Encoder Value	UInt32	4 Bytes	Read	Raw Value of Y Encoder
8	X Position	Float	4 Bytes	Read/Write	X Estimated Position, see below
9	Y Position	Float	4 Bytes	Read/Write	Y Estimated Position, see below
10	H Orientation	Float	4 Bytes	Read/Write	Estimated heading, see below
11	X Velocity	Float	4 Bytes	Read	Estimated X Velocity
12	Y Velocity	Float	4 Bytes	Read	Estimated Y Velocity
13	H Velocity	Float	4 Bytes	Read	Estimated Orientation Velocity
14	Ticks per mm	Float	4 Bytes	Read/Write	Encoder Ticks Per Linear mm
15	X Pod Offset	Float	4 Bytes	Read/Write	X Pod Offset from Tracking Point
16	Y Pod Offset	Float	4 Bytes	Read/Write	Y Pod Offset from Tracking Point
17	Yaw Offset	Float	4 Bytes	Read/Write	Yaw Scalar for Heading
18	Bulk Read	n/a	40 Bytes	Read	See Below

Device Status Register
This captures the status of the device with a bitwise operation. This allows this register to capture multiple simultaneous states.
Not Ready: 0
Ready: 1
Calibrating: 1<<1
X Pod Not Detected: 1<<2
Y Pod Not Detected: 1<<3

Device Control Register
This allows the user to send IMU Reset commands, reverse the encoder directions, and reset the position. It is a bitwise operation to keep it compact. It also includes control for factory IMU calibration. Do not set these bits.
Reset IMU: 1<<0
Reset IMU and Position: 1<<1
Set Y Encoder Reversed: 1<<2
Set Y Encoder Forward: 1<<3
Set X Encoder Reversed 1<<4
Set X Encoder Forward 1<<5
Gyro Start Calibration (do not use) 1<<6
Gyro End Calibration (do not use) 1<<7

Position Registers
If you read these registers, you read the estimated position of the robot. If you write these registers, you write a new position to the robot. Writing over these registers is how the setPosition functions work.

Bulk Read Register
This register allows you to read data normally stored in multiple registers with one I^2C read. This is intended to simplify the operation for users who need to read the position of the device every loop, but do not need to set/read configuration options every loop. All data in this register is read only.

Data	Data Type	Length
Device Status	UInt32	4 bytes
Loop Time	UInt32	4 bytes
X Encoder Value	UInt32	4 Bytes
Y Encoder Value	UInt32	4 Bytes
X Position	Float	4 Bytes
X Encoder Value	Float	4 Bytes
Y Encoder Value	Float	4 Bytes
X Position	Float	4 Bytes
Y Position	Float	4 Bytes
H Orientation	Float	4 Bytes
X Velocity	Float	4 Bytes
Y Velocity	Float	4 Bytes
H Velocity	Float	4 Bytes