

Creator Documentation

Creator Documentation

Exported on 07/26/2020

Table of Contents

1	Getting started	8
1.1	Installation & setup	8
1.1.1	Installation	8
1.1.2	Creating a setup	8
1.2	Hardware setup.....	12
1.2.1	Tags.....	12
1.2.2	Anchors.....	13
1.2.2.1	Placing the anchors.....	14
1.2.2.2	Calibration vs measuring.....	14
1.2.2.3	The ideal case.....	14
1.2.2.4	The realistic case.....	16
1.2.2.5	The rules explained	16
1.3	Floor plan setup	18
1.3.1	Introduction	18
1.3.2	Uploading the floor plan.....	19
1.3.3	Matching the floor plan.....	21
1.3.3.1	Scale	21
1.3.3.2	Rotation	21
1.3.3.3	Coordinate system	22
1.3.3.4	Transparency.....	22
1.4	Discover and autocalibrate	22
1.4.1	Introduction	22
1.4.2	Discovering your devices	23
1.4.3	Autocalibration	25
1.5	Positioning	30
1.5.1	Introduction	30
1.5.2	Positioning settings	32
1.5.3	UWB settings	35
1.6	Connect with MQTT.....	38
1.6.1	Introduction	38
1.6.2	Connect with the local stream	39
1.6.3	Connect through the cloud stream	39

2	Arduino	44
2.1	Downloads.....	44
2.2	Tutorial 1: Ready to range (Arduino).....	44
2.2.1	Ready to range	44
2.2.2	Plug and play.....	45
2.2.3	The code explained	46
2.2.3.1	Imports	46
2.2.3.2	Setup.....	46
2.2.3.3	Loop	48
2.2.3.4	ledControl.....	49
2.2.4	Remote operation	49
2.3	Tutorial 2: Ready to localize (Arduino).....	50
2.3.1	Ready to localize	50
2.3.2	Anchor setup and measurement.....	50
2.3.2.1	Anchor setup	50
2.3.2.2	Measurement	51
2.3.2.3	The Pozyx coordinate system.....	51
2.3.3	Plug and play.....	53
2.3.4	The code explained	54
2.3.4.1	Setup and manual calibration.....	54
2.3.4.2	Loop	56
2.3.5	Remote positioning.....	57
2.3.6	Extras	57
2.3.6.1	Printing the configuration result.....	57
2.3.6.2	Printing the error code.....	58
2.3.6.3	Adding sensor information	59
2.4	Tutorial 3: Orientation 3D (Arduino)	60
2.4.1	Downloads.....	60
2.4.2	Plug and play	61
2.4.3	Understanding the sensor data.....	62
2.4.4	The code explained	62
2.4.4.1	Imports and setup.....	62
2.4.4.2	Loop	63
2.4.5	What's next?	64

2.5	Tutorial 4: Multitag positioning (Arduino)	65
2.5.1	Multitag positioning.....	65
2.5.2	Plug and play.....	65
2.5.3	Code additions and changes	65
2.5.3.1	Anchor configuration	65
2.5.3.2	Positioning loop	66
2.5.4	Caveats and closing remarks.....	66
2.6	Chat room (Arduino)	67
2.6.1	The Arduino code explained.....	68
2.7	Configuration of the UWB parameters (Arduino)	70
2.7.1	The different UWB parameters.....	70
2.7.2	Doing it in code	72
2.8	Changing the network ID (Arduino).....	72
2.8.1	The code explained	72
2.8.1.1	Parameters	72
2.8.1.2	Changing the ID.....	72
2.8.2	Finishing up	73
2.9	Combination of cloud and Arduino.....	73
2.9.1	Introduction	73
2.9.2	Requirements.....	73
2.9.3	Here we go!.....	74
2.10	Troubleshoot basics (Arduino)	76
2.10.1	Overview	76
2.10.2	Local and remote device check.....	76
2.10.2.1	Not finding a remote device	77
2.10.3	Network check	78
2.10.4	Interpreting the LEDs	78
3	Python	80
3.1	Downloads.....	80
3.2	Tutorial 1: Ready to range (Python)	80
3.2.1	Ready to range	80
3.2.2	Plug and play.....	81
3.2.3	The code explained	82
3.2.3.1	Imports	82

3.2.3.2	Emulating the Arduino flow.....	82
3.2.3.3	Setup.....	83
3.2.3.4	Loop.....	83
3.2.3.5	ledControl.....	85
3.2.3.6	Remote operation	85
3.3	Tutorial 2: Ready to localize (Python)	86
3.3.1	Ready to localize	86
3.3.2	Anchor setup and measurement.....	86
3.3.2.1	Anchor setup	86
3.3.2.2	Measurement	87
3.3.2.3	The Pozyx coordinate system.....	87
3.3.3	Plug and play.....	89
3.3.4	The code explained.....	91
3.3.4.1	Setup and manual calibration.....	91
3.3.4.2	Loop.....	91
3.3.5	Remote positioning.....	92
3.3.6	Extras	93
3.3.6.1	Printing the configuration result.....	93
3.3.6.2	Printing the error code.....	93
3.3.6.3	Adding sensor information	94
3.4	Tutorial 3: Orientation 3D (Python).....	95
3.4.1	Downloads.....	95
3.4.2	Plug and play.....	96
3.4.3	Understanding the sensor data.....	97
3.4.4	The code explained.....	97
3.4.4.1	Imports and setup.....	97
3.4.4.2	Loop.....	98
3.4.5	What's next?	98
3.5	Tutorial 4: Multitag positioning (Python).....	99
3.5.1	Multitag positioning.....	99
3.5.2	Plug and play.....	99
3.5.3	Code additions and changes	99
3.5.3.1	Anchor configuration	99
3.5.3.2	Positioning loop	100
3.5.4	Caveats and closing remarks.....	100

3.6	Changing the network ID (Python)	100
3.6.1	The code explained	101
3.6.1.1	Parameters	101
3.6.1.2	Changing the ID	101
3.6.2	Finishing up	102
3.7	Troubleshoot basics (Python)	102
3.7.1	Overview	102
3.7.2	Local and remote device check	102
3.7.2.1	Not finding a remote device	103
3.7.3	Network check	103
3.7.4	Interpreting the LEDs	104

How can we help you?


1 Getting started

- [Installation & setup](#)(see page 8)
- [Hardware setup](#)(see page 12)
- [Floor plan setup](#)(see page 18)
- [Discover and autocalibrate](#)(see page 22)
- [Positioning](#)(see page 30)
- [Connect with MQTT](#)(see page 38)

1.1 Installation & setup

1.1.1 Installation

You can download the Pozyx Creator Controller [here](#)¹. You will find installers for Windows, MacOS and Linux AppImage, with additional instructions per platform.

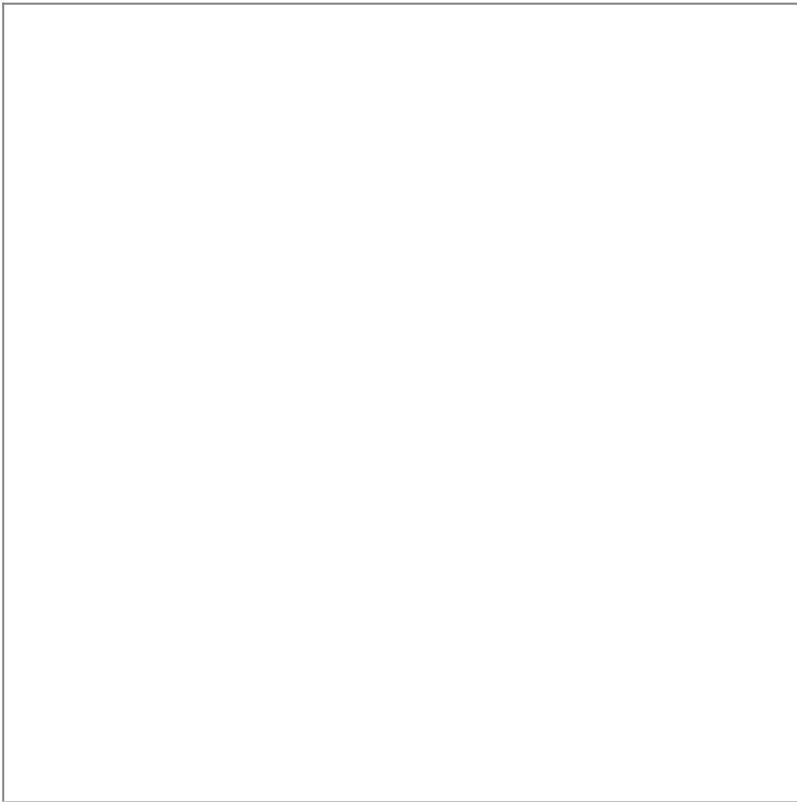
 If you bought your device before April 2018, this would be a good time to update the firmware on your devices by following the firmware update instructions. If you think your devices are very old, please contact our support first.

1.1.2 Creating a setup

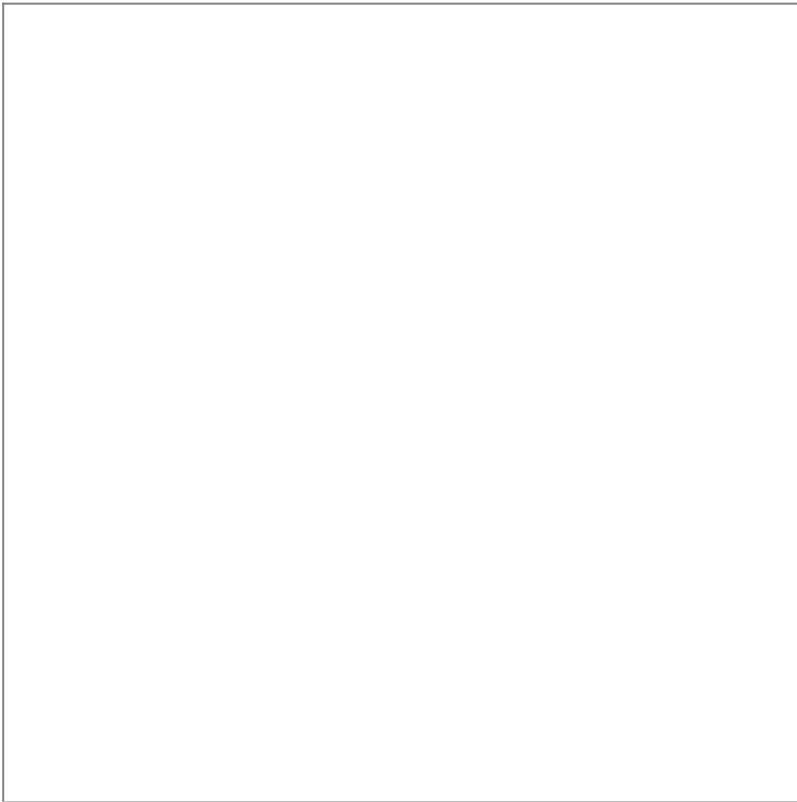
In this category we will follow the setup for Windows. For other platforms, the main difference is the appearance of the tray icon.

After installing the software, you can launch the Pozyx Creator Controller. The software will show itself as a tray application on your computer. As you never logged in before, you will be greeted by a login window.

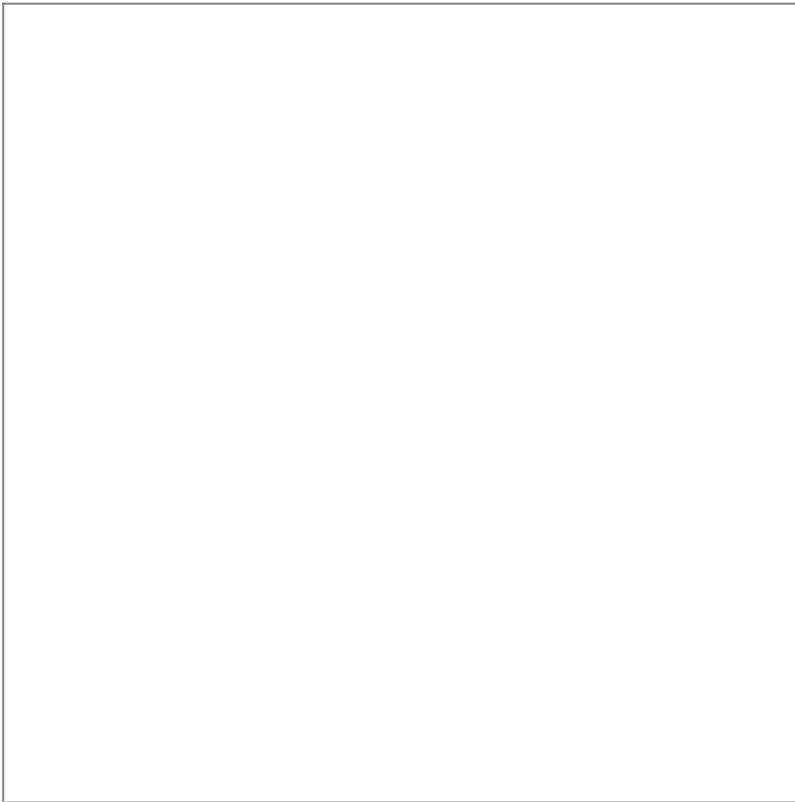
¹ <https://www.pozyx.io/documentation/pozyx-creator-controller>



If you have an existing account, you can log in with it here. If you don't, you can click on 'Create new account', which will take you to the registration window.



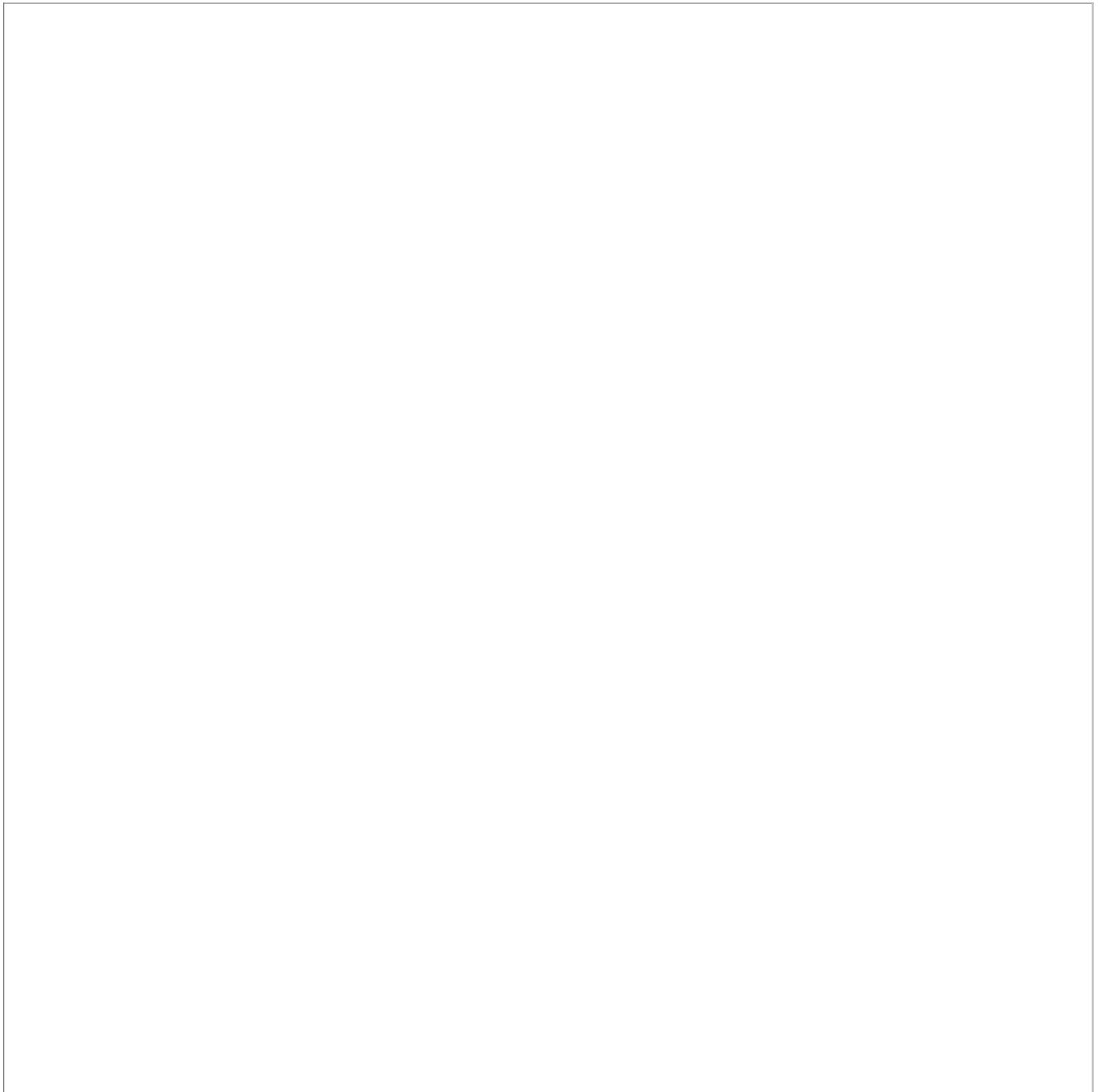
After either logging in or creating a new account, you're given the task of naming your setup. While it's tempting to name your setup "Test" or "Setup 1", names are important and a descriptive name such as "Office Prototype" or "Home Living Room" will help when you have multiple setups in the future. If you previously made setups, you can choose from those here as well.



After logging in creating/choosing a setup, the visualization will open. This is a standalone window that shows our [companion software](https://www.pozyx.io/product-info/companion-software)² which you can also see on <https://app.pozyx.io>. If you decide to use a browser, you will have to log in again with the same credentials and choose your setup.

You will be invited to follow a wizard to get acquainted with the interface. It's recommended that you follow this well. As shown in the image, both the connection status orb and button have to be green. This means your software is connected and authenticated successfully with the Pozyx cloud.

² <https://www.pozyx.io/product-info/companion-software>



You're now ready to set up the physical hardware!

1.2 Hardware setup

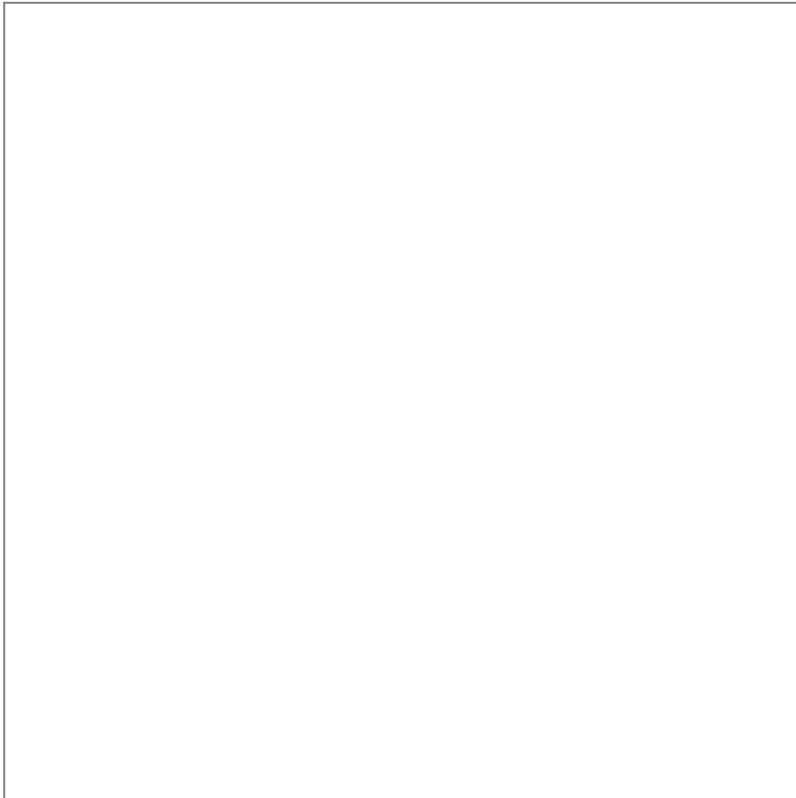
1.2.1 Tags

Tags are the devices that are positioned in a Pozyx setup. To perform positioning, they require anchors in their range to perform range measurements, from which a position is calculated. These typically move around, and can be powered using a small battery or power bank.

When working with the Creator Controller, one Pozyx device, be it a tag or anchor, has to be connected to the computer running the software. This device fulfills the role of master device, and is control of all UWB communications in the setup. You can always change which device you connect to your computer, but for a first setup it makes sense to connect one of your tags.

The other tags you have can simply be powered by a battery.

When you connect the tag to your computer, you can verify that it's connected via the tray application, as it will change its icon and its menu will state that a device is connected. If your tag does not get recognized, you may have to try a different USB cable or follow the detailed troubleshooting when you click 'No Pozyx device connected...' in the tray application.



1.2.2 Anchors

Anchors are Pozyx devices that are intended to never move and act as reference points for the tags and their positioning algorithm.

Installing the anchors and what to look out for is explained in detail in the anchor setup guide. With the companion software, you can easily auto-calibrate the anchors, so you don't have to worry about measuring the anchors' exact locations.

If you are done with setting up your hardware and have all devices powered up, you're ready to go to the next step: discovering your devices and auto-calibrating your anchors.

1.2.2.1 Placing the anchors

The Pozyx positioning system requires that at least four anchors are placed around/inside the area where you will be positioning. The accuracy of the system depends a lot on the quality of the setup, which should follow these rules of thumb:

1. Place the anchors high and in line-of-sight of the tag(s).
2. Spread the anchors around the area, never all on a straight line! Example: the corners of a room.
3. Keep the distance between two anchors in a range of 2 - 20 meters.
4. Keep the anchors at least 20 cm away from metal.
5. Place the anchors vertically with the antenna at the top or bottom.

Optional:

6. For 3D positioning: place the anchors at different heights.
7. If you want to perform auto-calibration, try to have line of sight between your anchors as well.

Before you install the anchors (with the provided Velcros or screws) on the walls or ceiling, it is usually a good idea to make a small sketch of the room, writing down the anchors' IDs and sketching where they will be placed. You can find the network ID as the hexadecimal number on the label adorning your anchor.

Remember that, for optimal antenna performance, it is recommended to place the anchors vertically with their antenna at the top, and to orient your tags that will be positioning vertically as well. Also make sure that no heavy metal objects are placed near the antenna, as this might degrade performance. We stress this again because it's just that important.

You can find a full explanation of every rule further below. We'll first go over why should either autocalibrate or measure your anchor coordinates.

1.2.2.2 Calibration vs measuring

There are two essential ways in figuring out your anchor's coordinates, and both have their advantages. To get started with positioning, we provide a world-class anchor autocalibration in our [companion software](#)³.

The resulting anchor positions should be accurate enough for most setups, providing accurate positions, and take away the difficulty and time necessary to measure out your anchor positions. For complex setups where it's hard to measure anchor coordinates, this is especially valuable.

Autocalibration is however not always the right solution. When you have an environment where the anchors don't have a line of sight connection with their neighbors, or are too far away, the autocalibration quality will be lower and so will the positioning accuracy. In these cases, you can measure out the anchor coordinates yourself, preferably with accurate equipment like a laser measurer or total station.

If you have an offline installation or don't want to work with the companion software, you will also need to measure the coordinates.

1.2.2.3 The ideal case

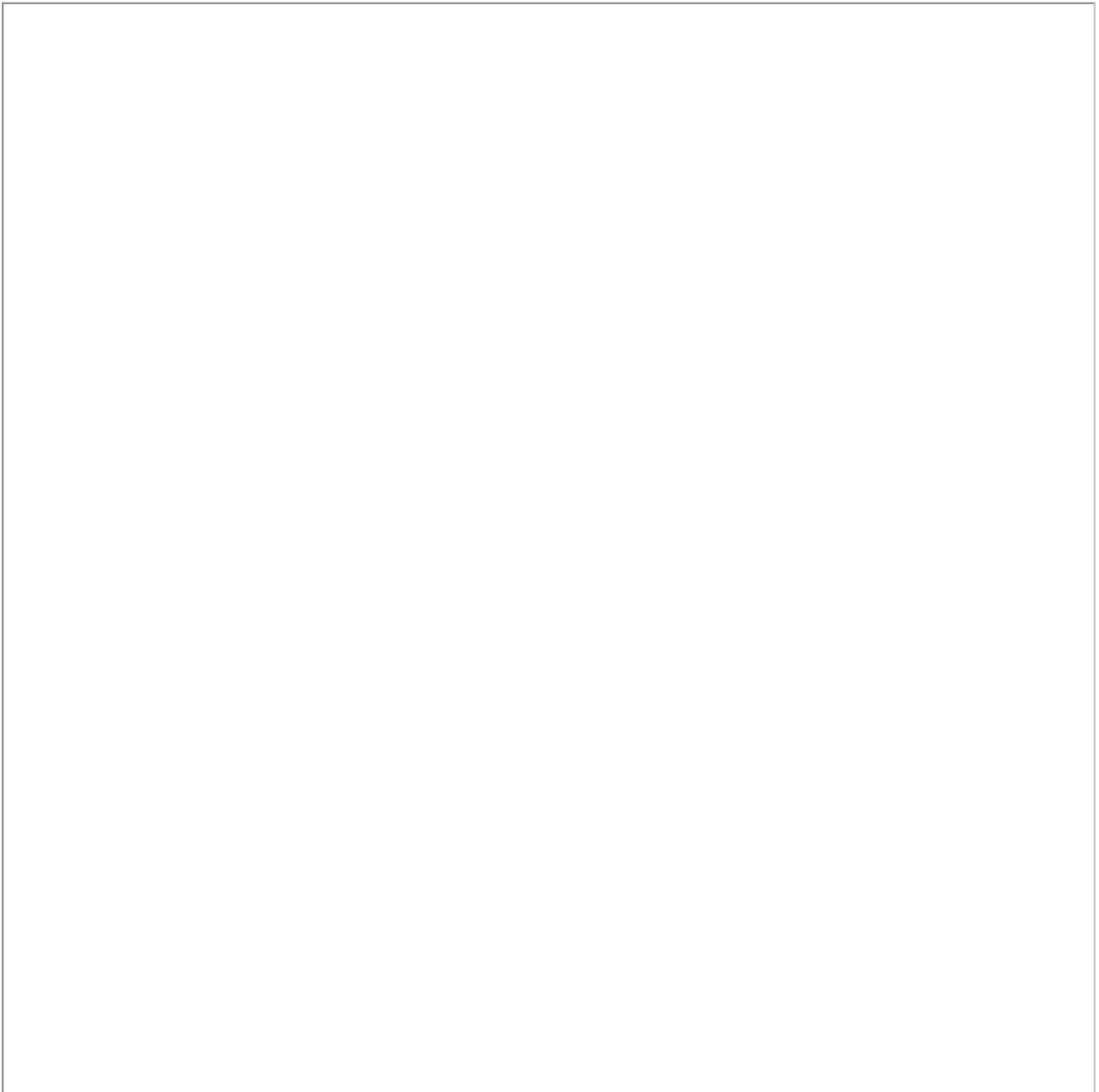
In an ideal case, you will be positioning in a single room that doesn't have too many objects inside of it, allowing for clear line of sight between the devices.

The image below illustrates the layout you should get when you follow the rules of thumb: an anchor in every corner so that every point inside the room is inside the anchor's bounding box, and every anchor can have optimal transmission.

³ <https://www.pozyx.io/product-info/companion-software>

The anchors on one diagonal should be near the ceiling, while the anchors on the other diagonal should be near the ground. This way, the anchors do not form a 3D plane and the positions can't be mirrored (see rule 6). If you'd choose to manually measure your anchor coordinates, a setup like this makes defining coordinates a lot easier as well.

On the right, the anchors are thrown on the ground and the tag is being positioned outside of the anchor's bounding box. Because of to the worse antenna transmission, proximity of the anchors and positioning the tag outside of the anchor's enclosing area, the positions here will be considerably worse than the setup on the left.



1.2.2.4 The realistic case

Only few setups have an ideal environment and layout as below. When tracking over a large area, you will have anchors inside of the area to have a better coverage.

Especially when tracking in multiple rooms or with many (metal) obstacles, it's important to try to have 4 anchors within range and in line of sight of the tag in all areas where you want to perform accurate positioning.

1.2.2.5 The rules explained

Rule 1: Place the anchors high and in line-of-sight of the user

The first rule is straightforward: placing the anchor high (on the ceiling or on the walls) increases the chance of receiving a good signal because there are less obstructions. Obstructions generally have a negative influence on the accuracy of the range measurements which has a direct effect on the positioning accuracy.

Rule 2: Spread the anchors around the area, never all on a straight line!

For range-based systems, a single range measurements will only give information in a single direction. This direction is exactly the direction from the user to the anchor. Because of this, it is best to spread the anchors such that they cover all directions. If the anchors are all on a straight line, the positioning error will be very large. This can be seen on the image below. You can see that a small change in radius (for example due to noise), will result in a very large change in the position of the intersection(s). In other words, the error on the range measurements is amplified! This is the same principle as in GPS, where it is called the geometric dilution of precision (GDOP).

Rule 3: Keep the distance between two anchors in a range of 2 - 20 meters

To make sure you have the best coverage and accuracy (see Rule 2), it's important to spread out your anchors as much as possible. On the default UWB settings, a maximum range between the anchors of 20 meters is recommended so that your tag will always be in range of your anchors.

Additionally, the ranging measurement algorithm performs better when the devices are further apart, so it's best to keep some distance between the tag and anchors for improved accuracy. This is important for anchors as well if you want to perform autocalibration.

Rule 4: Keep the anchors 20 cm away from metal

When you place an anchor directly on a metal plate, the metal will affect the antenna behavior. It is advised to keep a distance of 20cm clear from the antenna with metal. The antenna itself is a metal conductor that is carefully designed to radiate on the UWB frequencies. Any metal nearby will reduce the antenna's efficiency or will make the antenna less omni-directional and less predictable. This rule is in fact true for every wireless system.

Rule 5: Place the anchors vertically with the antenna at the top or bottom

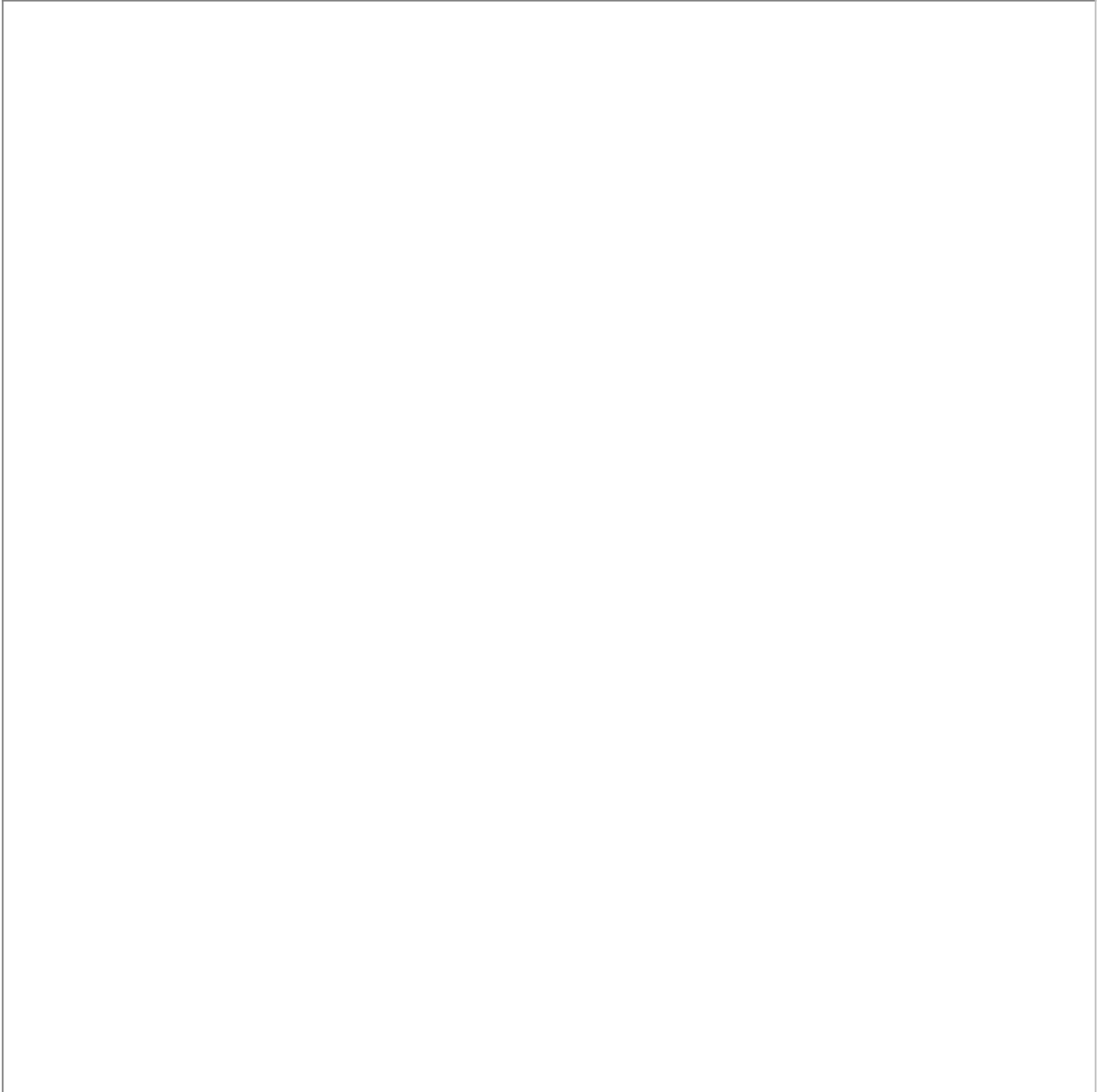
The Pozyx system uses wireless UWB signals for positioning and this requires an antenna. However, it is physically impossible to create an antenna that performs good in all directions. The monopole antenna (the white chip) on the Pozyx devices radiates omni-directionally in the xz-plane, but doesn't radiate as well along the y-axis (see figure). To make sure that we have the best possible reception, it is recommended to place the anchor vertically. Note that the same is true for the Pozyx tags. The ranging algorithm inside the tag is also optimized for this orientation.

Rule 6: For 3D positioning: place the anchors at different heights

The anchors need to be placed at different heights to be able to discern an accurate z coordinate, this follows the same principle as Rule 2, but applies to the z-axis while Rule 2 applied to the x and y accuracy.

Note that in a typical room, with line of sight considerations, the anchors can be placed between 1 and 2 meters height and the z-accuracy may not match the x and y accuracy.


It is also important that the anchors do not lay in the same 3D plane, as this creates a mirror plane for the tag's position, causing the tag's position to jump around to its real and mirrored positions. This is visualized on the right side below.



Rule 7: If you want to perform auto-calibration, have line of sight between your anchors

The autocalibration relies on range measurements between the anchors. If the anchor can not accurately range with each other, or don't have enough anchors in range to have enough measurements, the autocalibration results will not be trustworthy. After the autocalibration in the companion software, you can visually verify whether the result seems correct. Additionally, the software will inform you about the quality of the autocalibration.


1.3 Floor plan setup

 If you have a floor plan you can use with the setup, great! It's optional, but fully recommended to set it up. If you do not, you can always upload and configure your floor plan later.

1.3.1 Introduction

Floor plans differ in detail and size, but their most important properties are accuracy and scale. If you don't have a floor plan of the space you'll be positioning in, it's vital that the floor plan you will create has proper dimensions.

As an example, this is the floor plan of the Pozyx office. You can clearly see the different rooms, and the furniture details add a nice touch to compare the tags' positions with.

 If you bought your device before April 2018, this would be a good time to update the firmware on your devices by following the [firmware update instructions](https://www.pozyx.io/products-and-services/developer-tag/firmware)⁴. If you think your devices are very old, please contact our support first.

⁴ <https://www.pozyx.io/products-and-services/developer-tag/firmware>



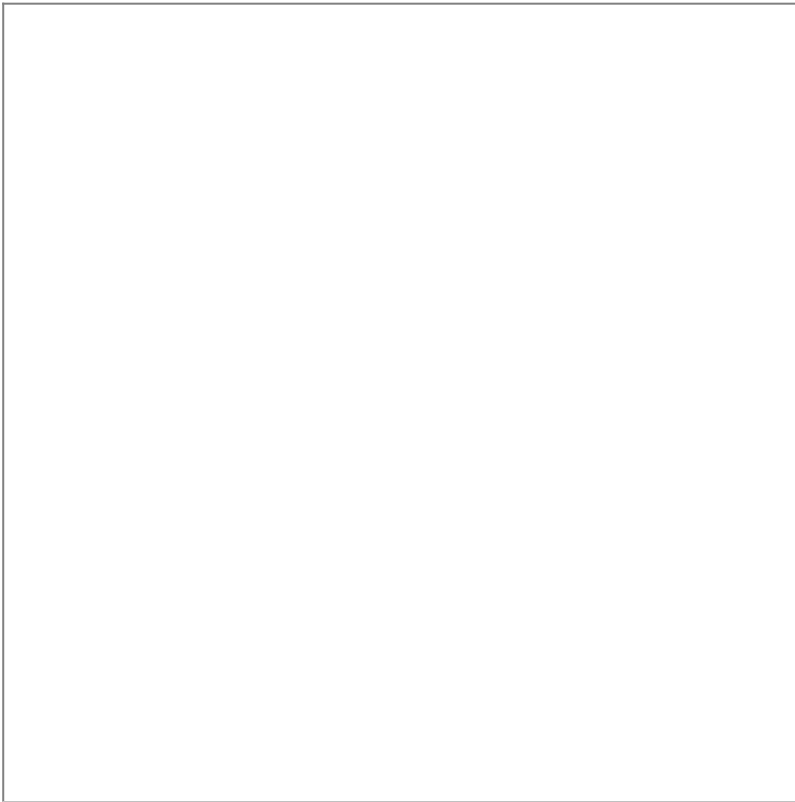
All distance measurements necessary to create this floor plan have been made with a laser measurer, and you can see the real life dimensions in the floor plan at the bottom of the image. We will be using this to scale the floor plan to match the coordinate system in the visualization, so it's very important to have at least one reference distance on the floor plan.

1.3.2 Uploading the floor plan

To configure the floor plan in the companion software, please navigate to the 'Floor plan' section of the Setup page, found under the Planning category.



You can upload any JPEG, PNG or GIF in the companion software. In the tutorial setup, we will upload and configure the Pozyx office floor plan.



1.3.3 Matching the floor plan

After uploading the floor plan image, you need to configure it have the same dimensions as the visualization. When you've uploaded the image, it will be pretty small and will likely need to be scaled and repositioned. After the floor plan has been configured the way you want, and you already have anchors set up, you can finely match the anchors against the floor plan in the 'Anchor coordinates' window.

It's very recommended to watch the video to see the process in action.

1.3.3.1 Scale

By far the most important operation is scaling the floorplan. For this, you need to mark two points on the floorplan and scale the floorplan using one of two methods.

- **Distance:** this is the easiest method, and needs you to know the distance between the two points you marked. After filling in the distance and pressing scale, you'll see that the two points will now be this distance apart, and the floor plan will be scaled to match this distance.
- **Coordinates:** this second method is great when you know two of the coordinates on the map. After filling in the coordinates of both points, the floor plan will not only have been scaled, but also rotated to match the coordinates. If you have pre-existing coordinates in which you want to use Pozyx, you should use this method.

1.3.3.2 Rotation

If you need to rotate the floor plan, you can do this in the third step. You can manually enter an angle, or rotate the floorplan using the handles.

1.3.3.3 Coordinate system

The coordinate system is editable, but it's important to know why. Not all floor plans are made equal, and some have a different coordinate system. This way, instead of flipping the floor plan and possibly flipping the text on it, you can change the coordinate system to match the one in your floorplan.

For most floor plans, the default coordinate system will be the right one.

1.3.3.4 Transparency

If you have a floorplan that limits the view of the Pozyx visualization by, for example, being the same color as ranges, anchors, or tags, it may be nice to lower the transparency.

1.4 Discover and autocalibrate

1.4.1 Introduction

With both your software and hardware installed, it's time to connect the two. If by chance you don't have the anchors set up, please do so as instructed here.

In this section, we'll discover all the Pozyx devices in range and let the anchors autocalibrate. For these steps, we're using the 'Anchor coordinates' section of the Setup, which you can find under Rollout.



1.4.2 Discovering your devices

The Pozyx connected to your computer can perform a ‘discovery’ broadcast which will find all devices in range. You can give this command in the Anchor coordinates page.

There are two options:

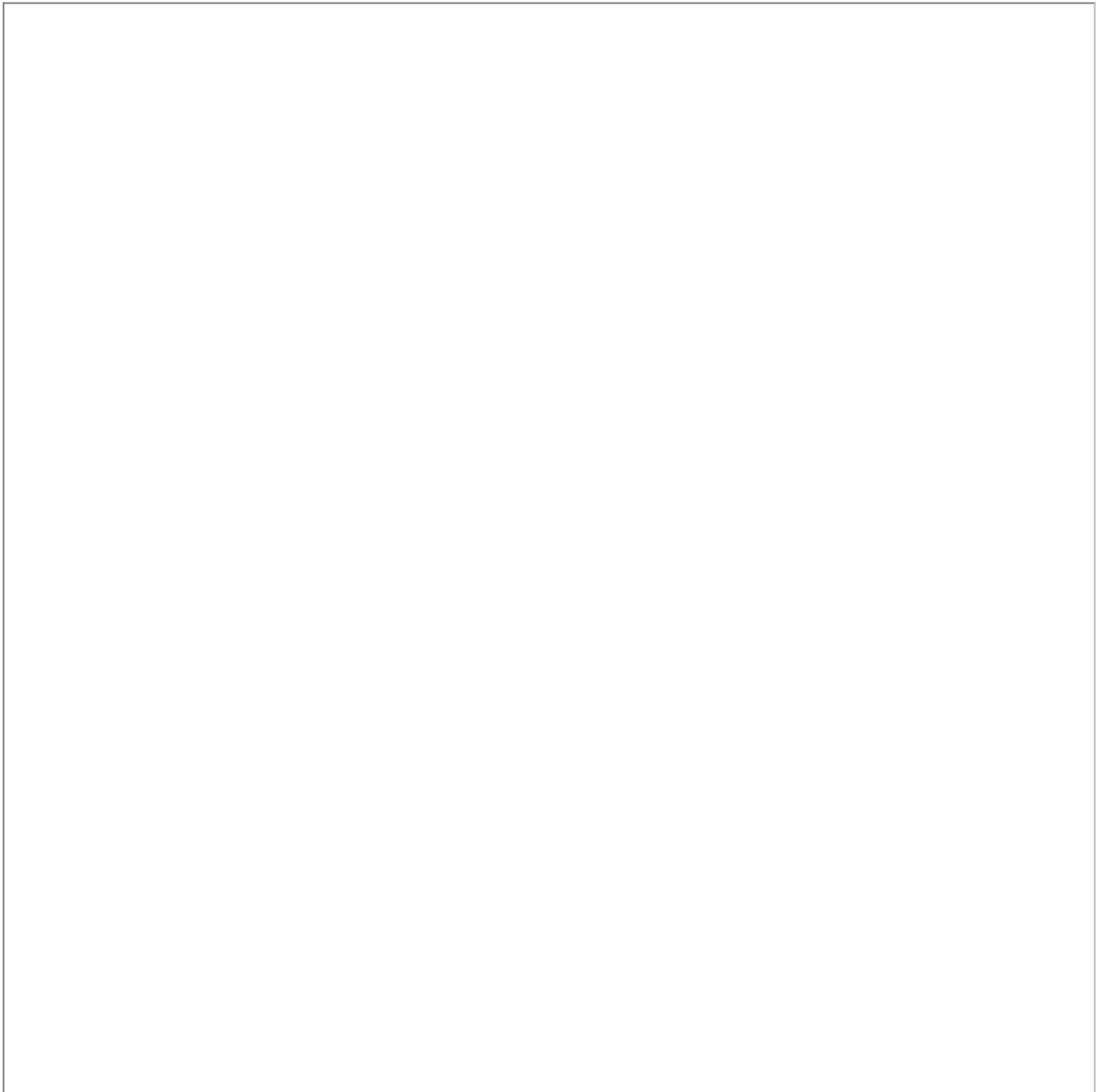
- Quick discovery: The quick discovery performs a discovery on the locally configured UWB settings and will find devices on those settings. It typically takes around 2 to 3 seconds.
- Full discovery: the full discovery goes over all the possible UWB settings and is intended to find Pozyx devices on other UWB settings. It takes about 10 seconds.

The full discovery is recommended if you don’t find all your devices with the quick discovery.



If all goes well, you will discover the four anchors you installed and at least one tag. If you don't pick up on all your devices, check whether they are in range/line of sight and are powered on. If they are, try a full discovery.

The anchors should show up as a list with empty coordinates, with the Z coordinate 0 by default. In the next step, we will auto-calibrate the anchor's positions.



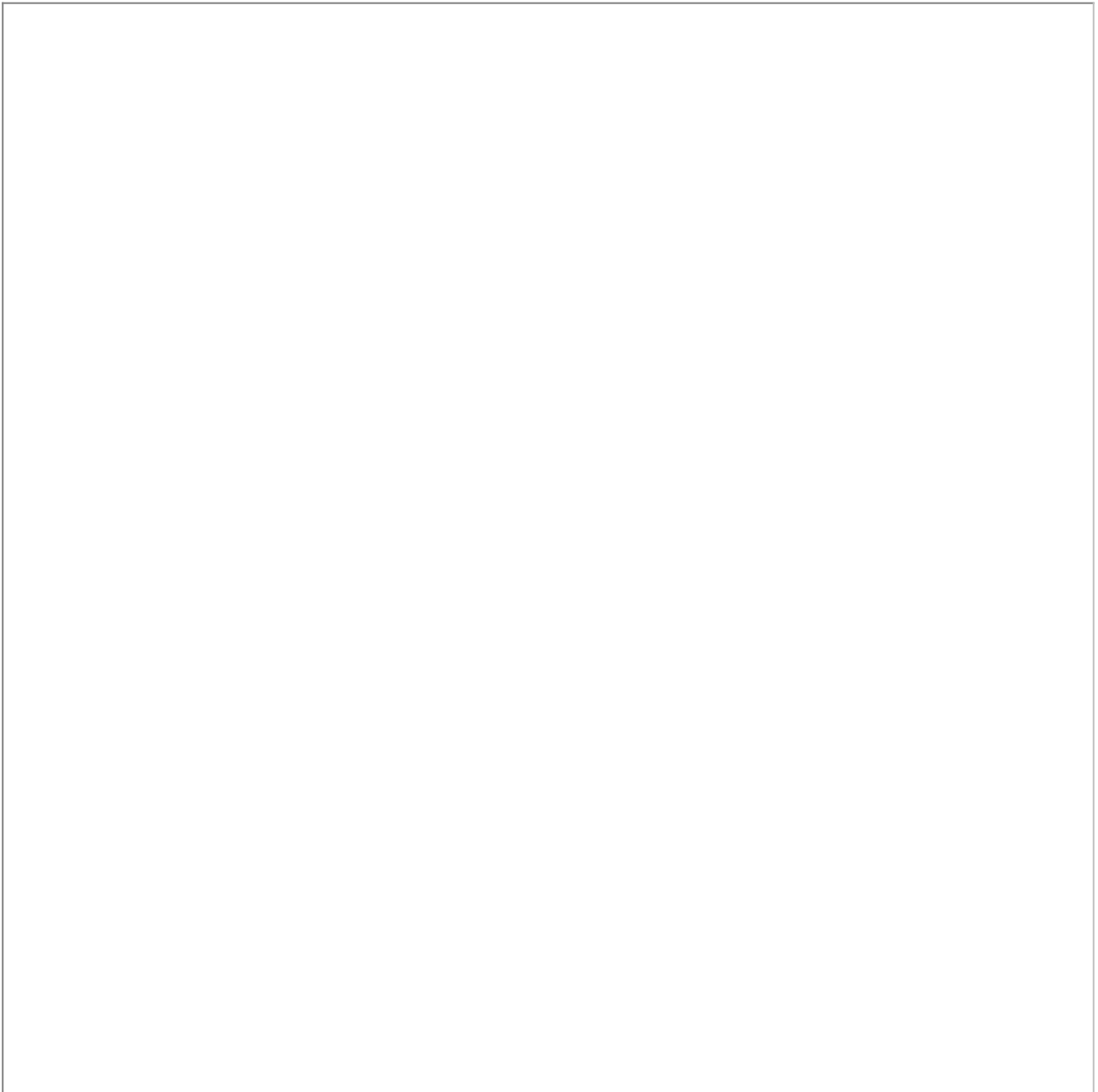
1.4.3 Autocalibration

It's tempting to push that 'Calibrate XY-coordinates' button right away, but we need to do one important thing first: setting the heights of the anchors. By default, the anchor height is fixed to 0 as a default value.

In the tutorial setup, all anchors are on a height of 2.4 meters, or 2400 mm. We'll fill this in. The lock icon after the coordinate means it won't change during the autocalibration.



After setting your anchor heights, it's finally time to press that 'Calibrate XY-coordinates' button! You will see that the calibration is running.

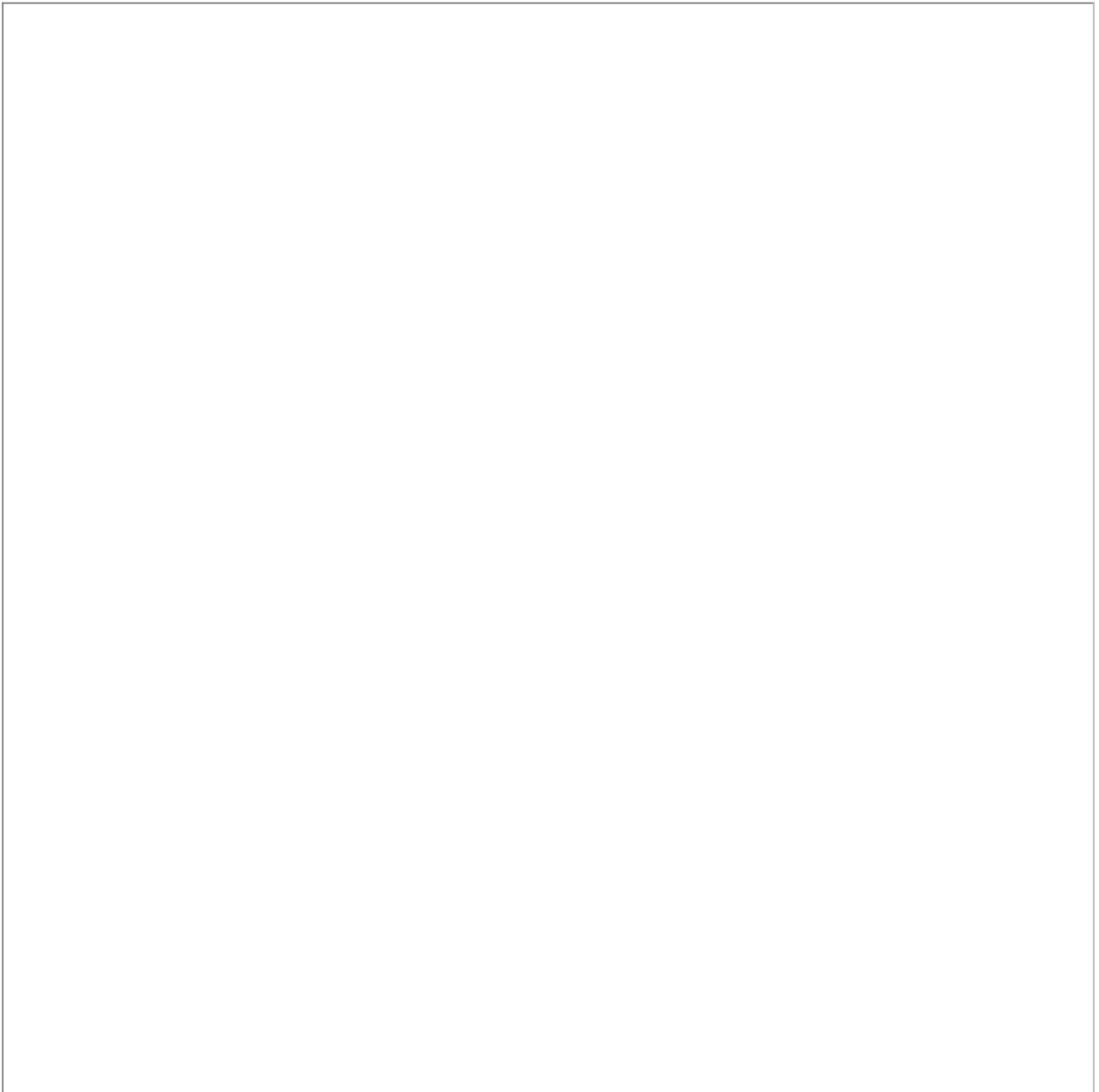


After a few seconds, the autocalibration will finish and you will see that not only your anchors have coordinates, but they're also visible on the visualisation on the right. If you don't see your anchors, press one of the crosshair icons next to the anchors. This will center the view on that anchor.

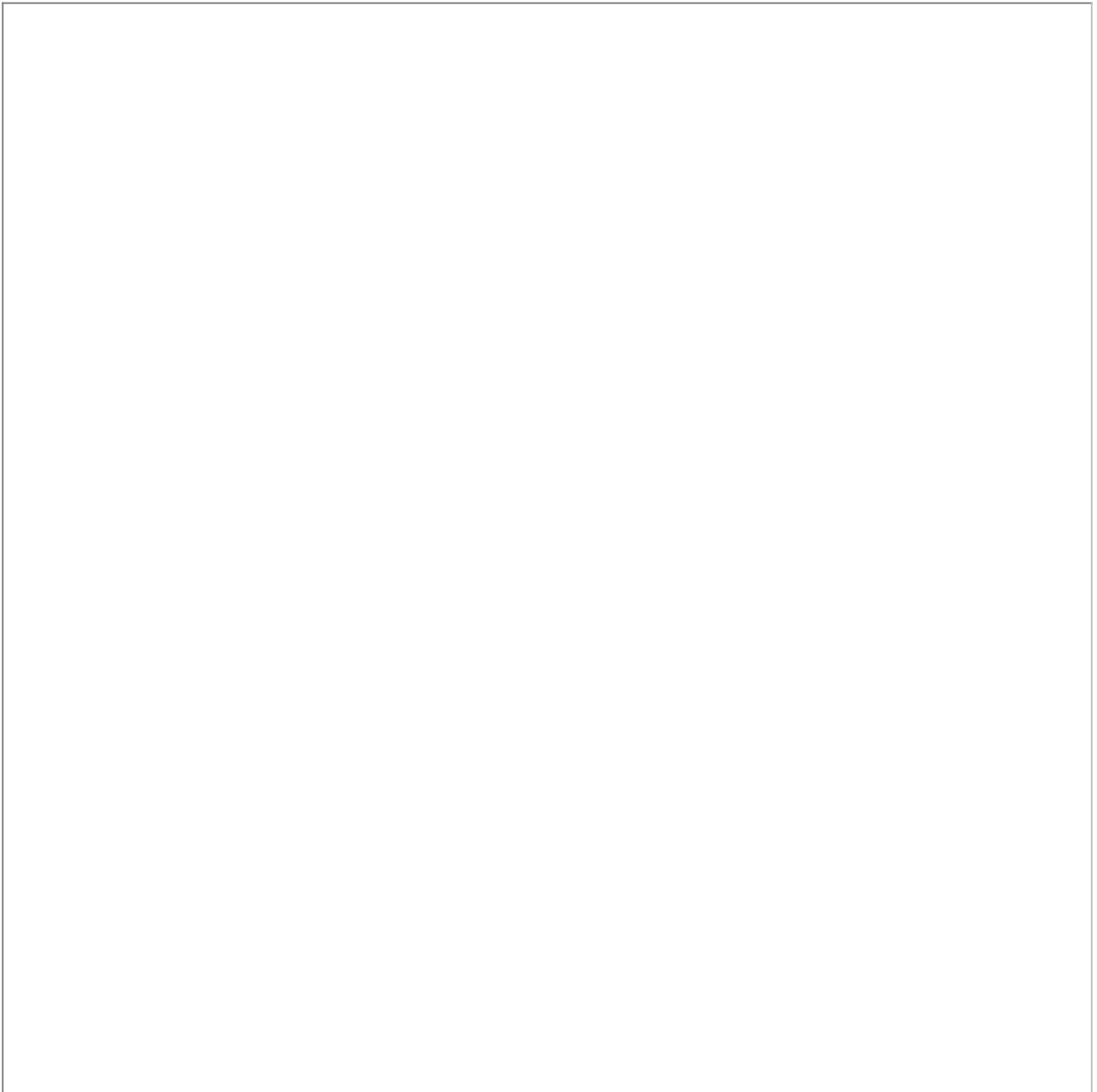
Additional information you can see here is the range measurement's distance and quality between any pair of anchors, and how many neighbors each anchor has. It's recommended to have at least 3 neighbors for every anchor when you're autocalibrating.

You can also rotate and drag the anchors around here, just don't forget to press 'Save transformation' when you're done.

It's not possible to scale the anchors, as this would not keep the anchor positions accurate relative to one another. Rotation and translation are non-destructive operations for the relative positions.



If you have uploaded a floor plan, you can fit the anchors on the floor plan now, saving you the trouble of doing this later. Below you can see the anchors fitted against a floor plan of our office, providing an instant visual confirmation of whether these positions are accurate.



Now, it's time to start positioning!

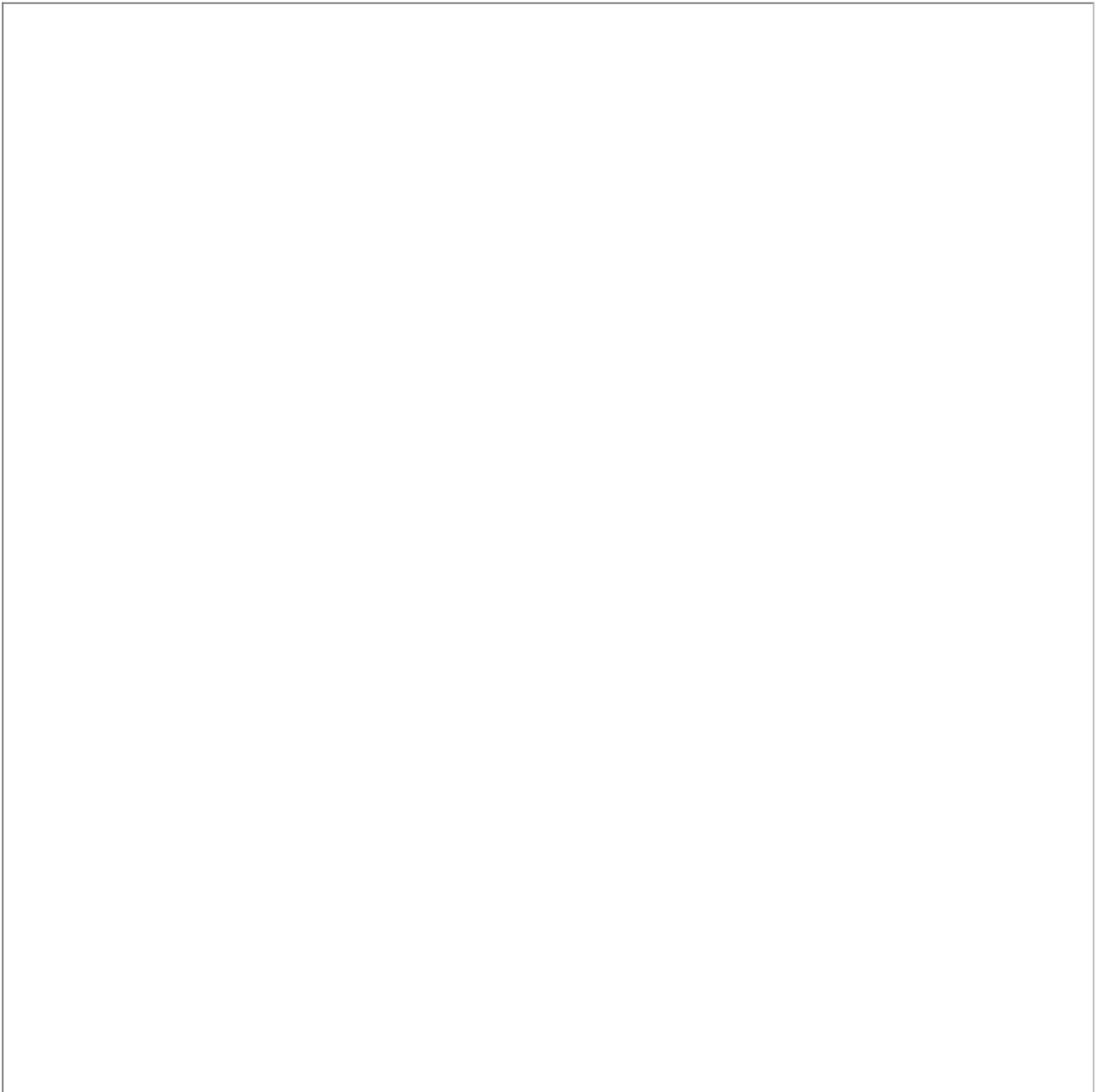
1.5 Positioning

1.5.1 Introduction

Once your anchors have coordinates, the positioning will start automatically. You can see this in the [visualization page](#)⁵.

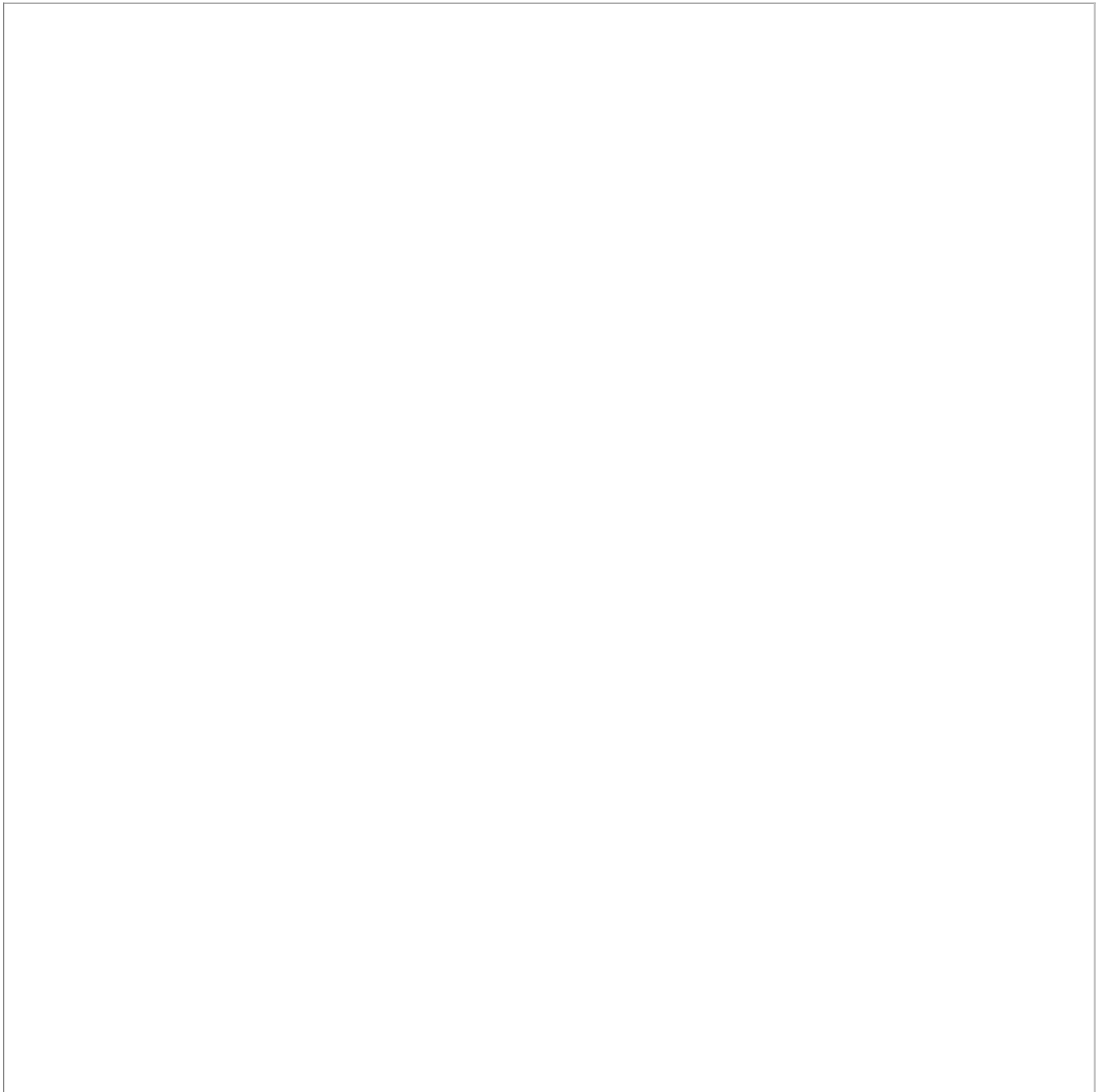
When clicking on the buttons showing the amount of working tags or anchors, you will see a list of those respective devices. Each ID is also clickable to provide more information about that particular device's settings.

⁵ <https://app.pozyx.io>



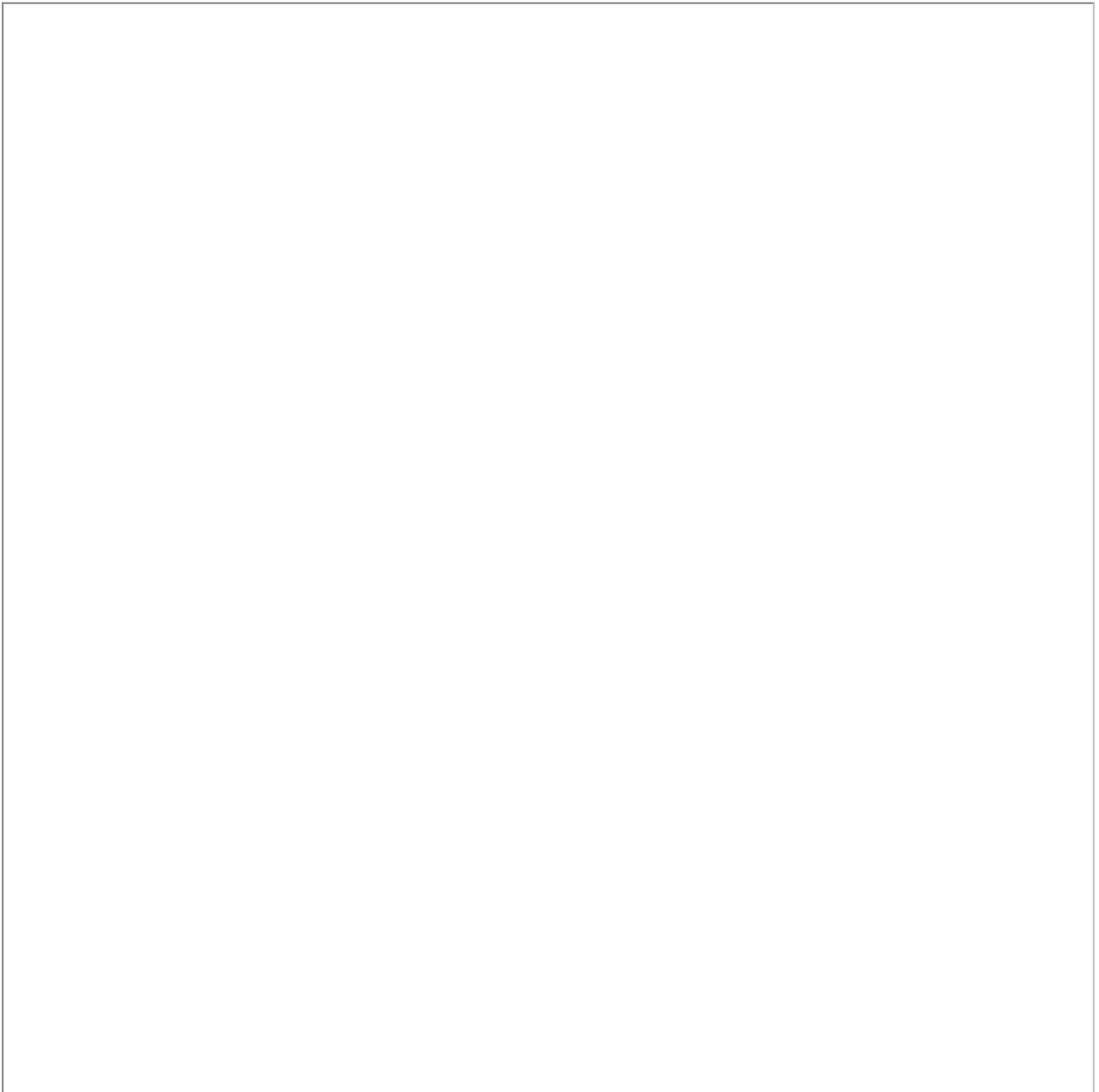
For tags, you can see the update rate, the current coordinates, the UWB settings... but also the firmware and hardware versions on the device. The selected tag will also be highlighted, and other tags will be shown less bright.

This works for anchors as well. When pressing the "Actions" foldout, you can also choose to forget the individual device, removing it from the list.



1.5.2 Positioning settings

Not every installation is the same, and sometimes you will want to tweak some of the positioning settings to get the best result. You can navigate to both positioning settings and UWB settings in the ‘Settings’ page of the companion software.

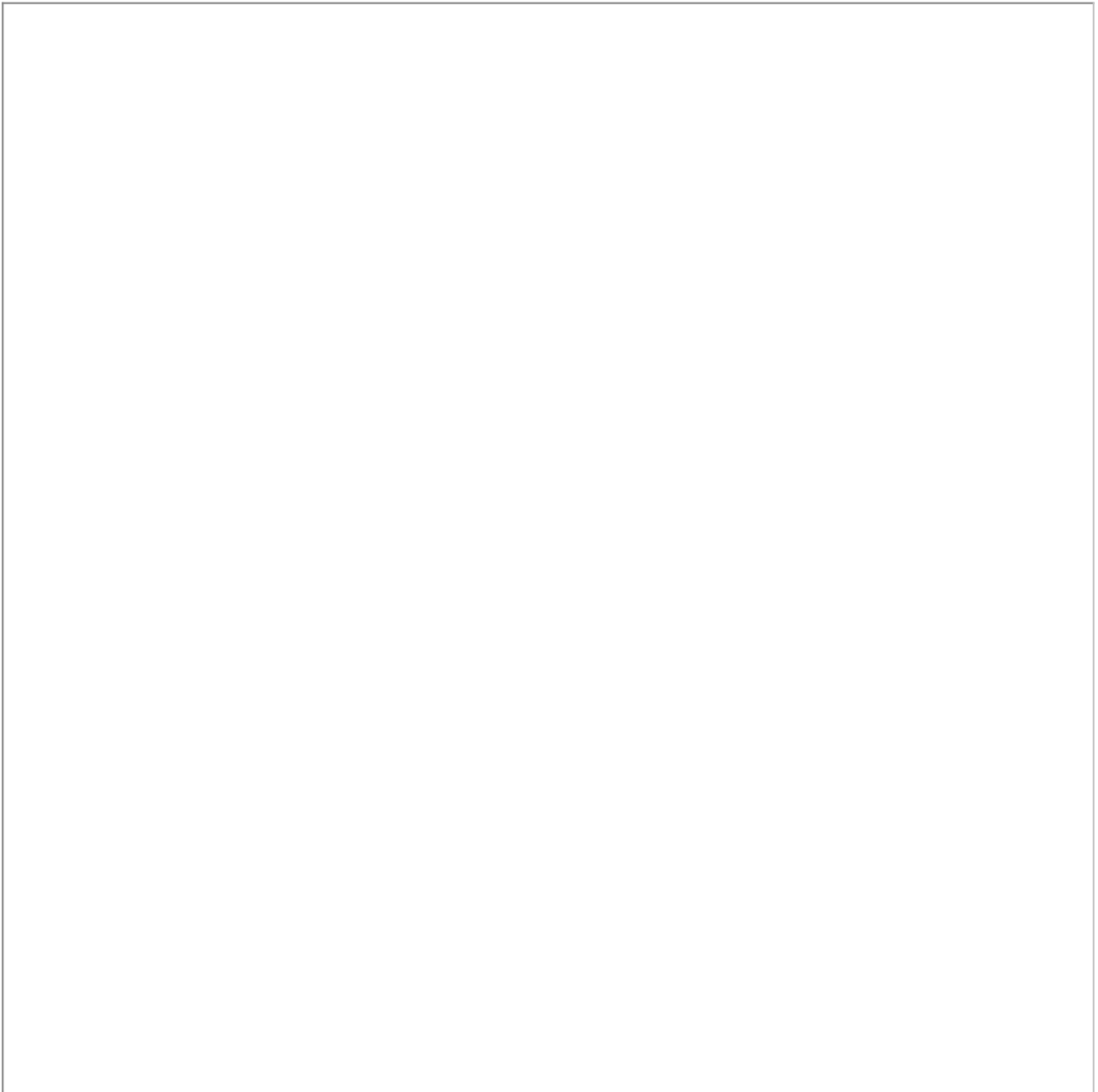


The positioning settings offer options to configure the operation of the positioning algorithm. The default settings should suffice for a lot of setups, offering a good update rate and good positioning performance.

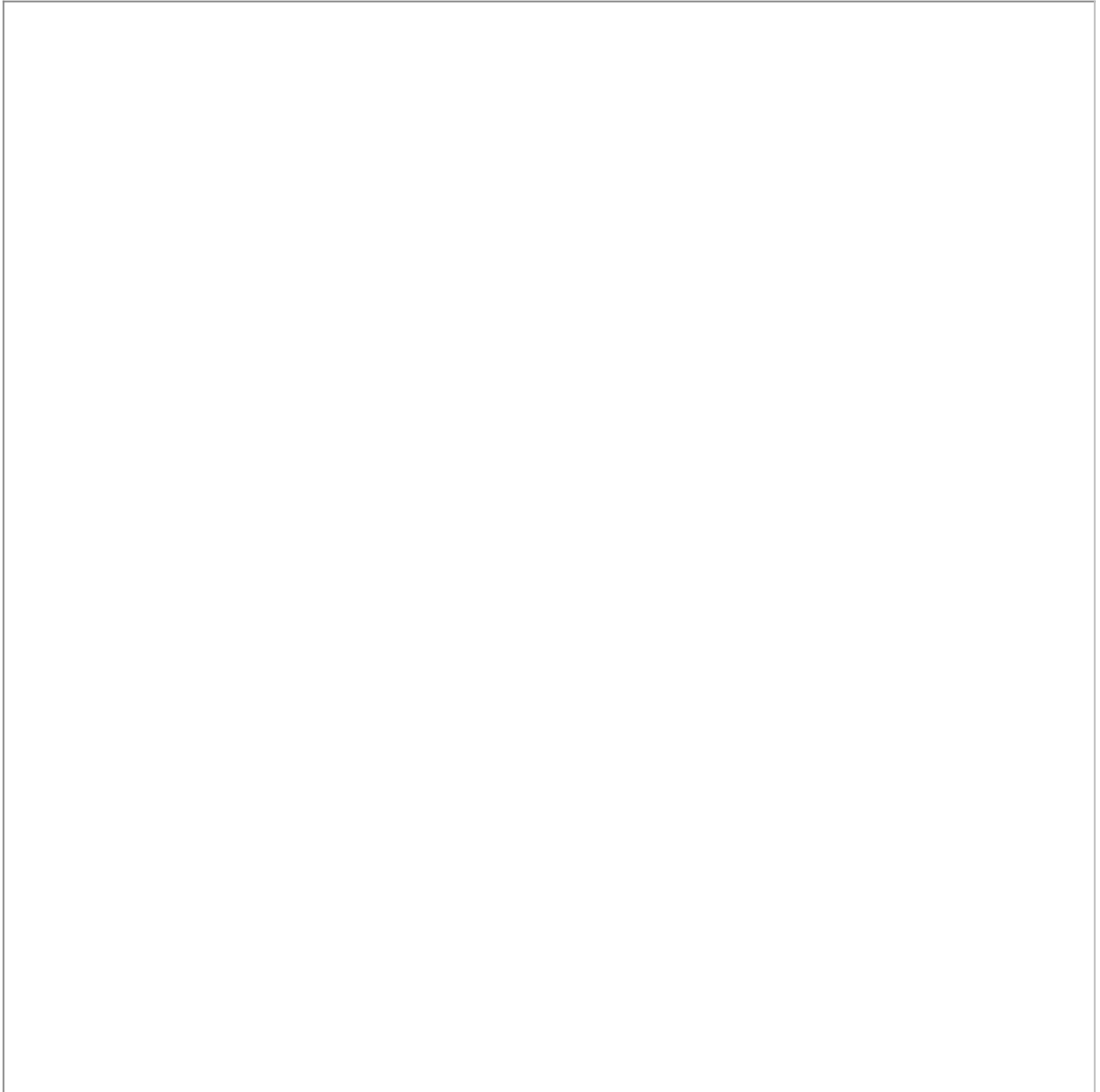
As you can see below, hovering over an option will show you detailed information about that options. Hovering over the different options gives detailed information about the option.

You can also configure a filter that will be applied to the positions. This way, you don't need to do this in post-processing. In general, increasing the filter strength slider will make the positioning smoother, but will also increase the delay on the position.

The sensor data allows you to configure which extra tag sensor data should be present in the positioning stream.




An important new feature is ‘Off-board’ positioning. When positioning this way, the tag will calculate its ranges with the anchors but the final position will be calculated in the Creator Controller. This should both improve the update rate and positioning accuracy, especially in the z-direction. This is a new feature, and is not enabled by default to prevent regressions.



1.5.3 UWB settings

The UWB settings also have a large impact on the positioning and the update rates that can be achieved. However, playing with the UWB settings is considered more advanced for the simple reason that all devices in the system must be on the same UWB settings to work together.

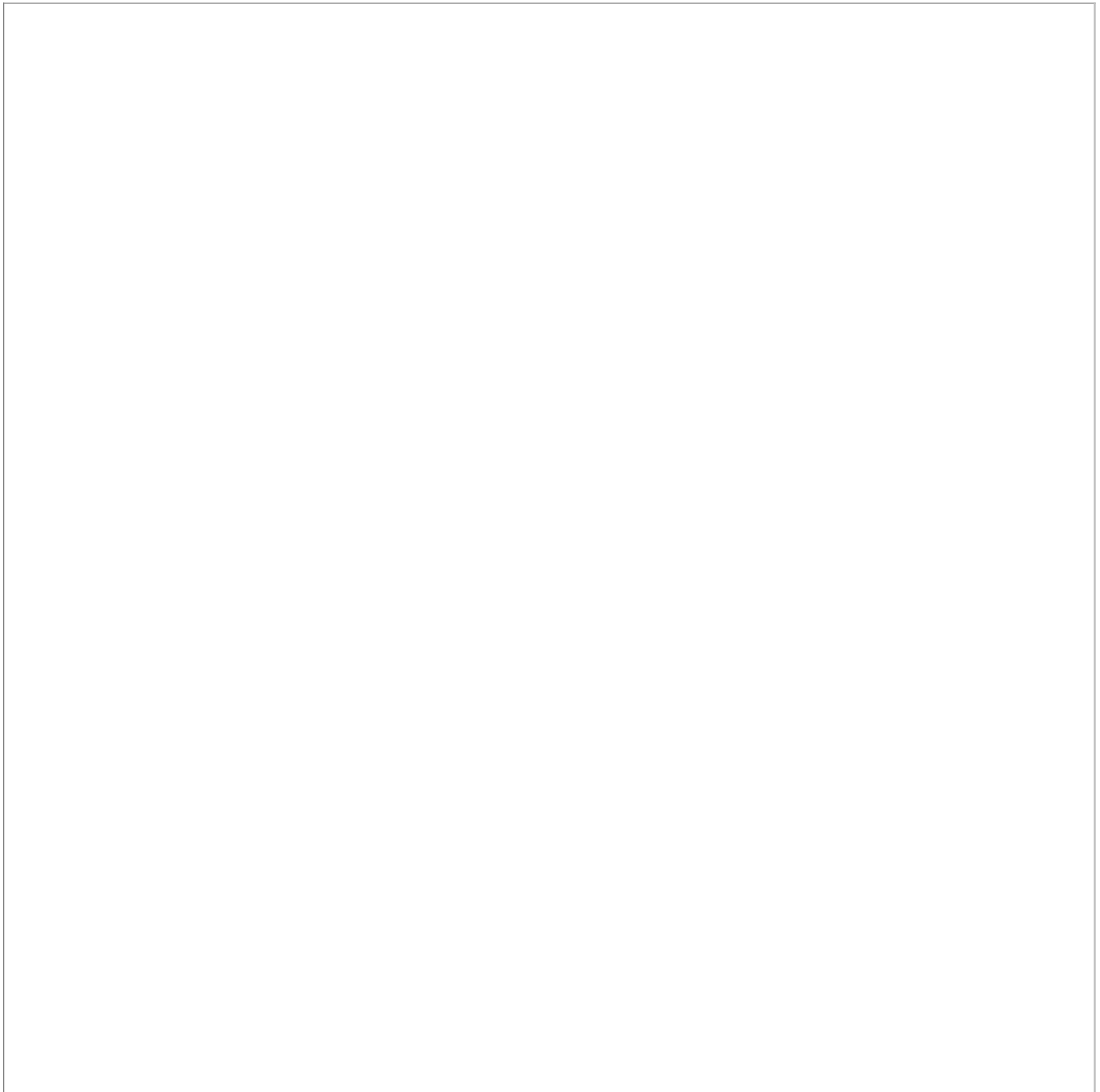
 Saving the settings will try to save the UWB settings on all tags and anchors within range that have been discovered and are in use. This is not always possible. The settings may result in a different radio range and consequently, some devices may be in range for one combination of UWB settings, but not for another.

For larger setups, it may be necessary to move around to update all devices and to verify their settings using discoveries.

If you have old or new devices on different UWB settings than you configured, or can't find your devices in general, performing a 'Full discovery' is recommended.

For each setting the system again provides useful tooltips on its effect, but they're listed in more detail here below.

- **Channel:** the UWB channel. The pozyx device can use 6 independent UWB channels. Devices on different UWB channels cannot communicate and do not interfere with each other. In general, lower frequencies (i.e. lower channel numbers) also result in an increased communication range. The antenna on the developer devices works has the longest range on channel 2.
Data bitrate: the UWB bitrate. Three possible settings are possible: 110kbit/sec, 850kbit/sec and 6.81Mbit/sec. A higher bitrate will result in shorter messages and thus faster communication. However, this comes at the expense of a reduced operating range. The effect of the bitrate on the duration of ranging or positioning is shown in the figures below.
- **Pulse repetition frequency (PRF):** the UWB pulse repetition frequency. Two possible settings are possible: 16MHz or 64MHz. This settings has little effect on the communication rate. However, when you want two setups on the same UWB channel, changing this setting between the two will reduce interference.
- **Preamble length:** the UWB preamble length. This setting has 7 different options: 64, 128, 256, 512, 1024 or 2048 symbols. A shorter preamble length results in shorter messages and thus faster communication. However, this again comes at the expense of a reduced operating range.
- **Tx gain:** the UWB transmit power. The power can be set between 0 and 33 dB, and each channel has a default value. The maximum transmit power you can legally configure your devices with depends on your country.



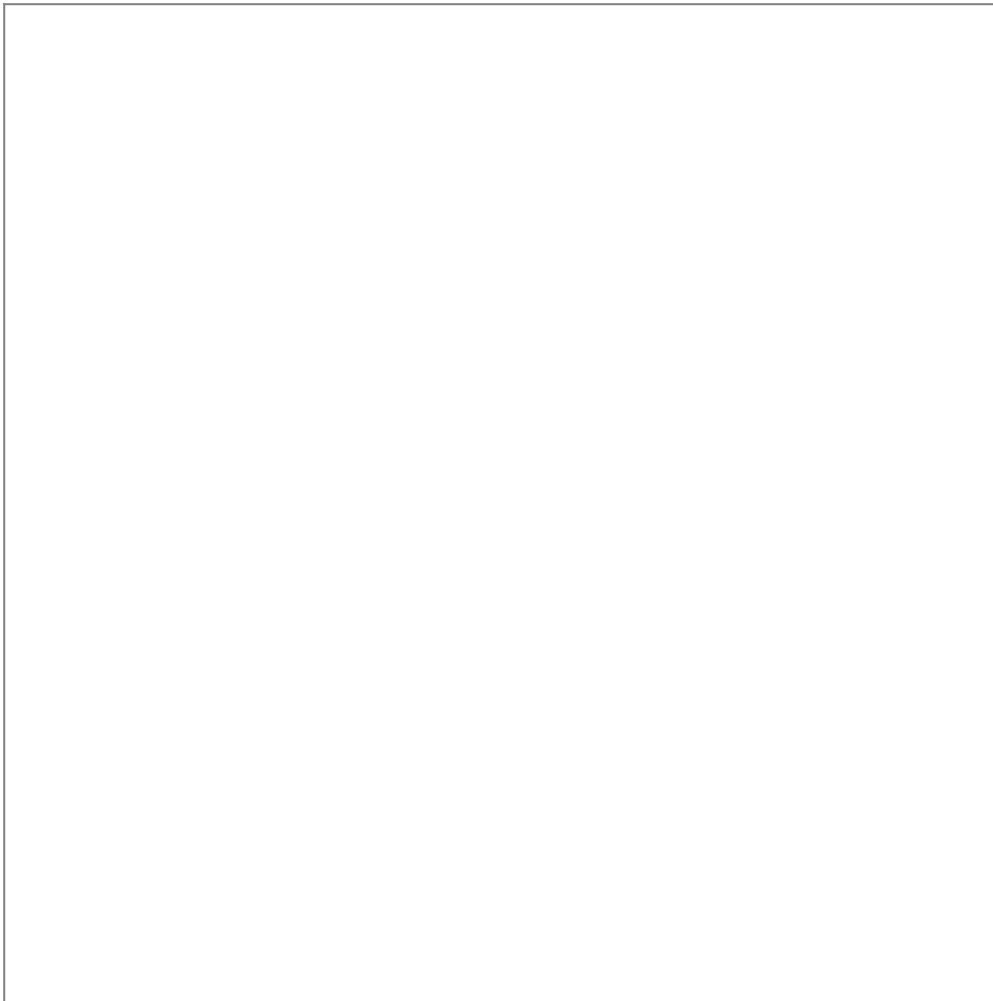
Now that your tags are positioning, you can easily integrate the Pozyx position data in your own applications using our MQTT streams. The next section explains how to do this for both a local and a cloud-based stream.

1.6 Connect with MQTT

1.6.1 Introduction

With the full system set up and the positioning working, we would like to extract the data out of the system to do something with it. This can be done very easily using the popular [MQTT protocol](http://mqtt.org/)⁶. MQTT is an IoT protocol that allows to capture sensor data in a publish/subscribe method. By subscribing to a topic with the positions, you will get the newest positions as soon as they are available.

By default, only positions are embedded in the MQTT packet. However, it is possible to embed more sensor data from the tag in the packet. This can easily be configured in the settings page for positioning under settings > positioning. With the dropdown, it is possible to select all the different sensors.



⁶ <http://mqtt.org/>

1.6.2 Connect with the local stream

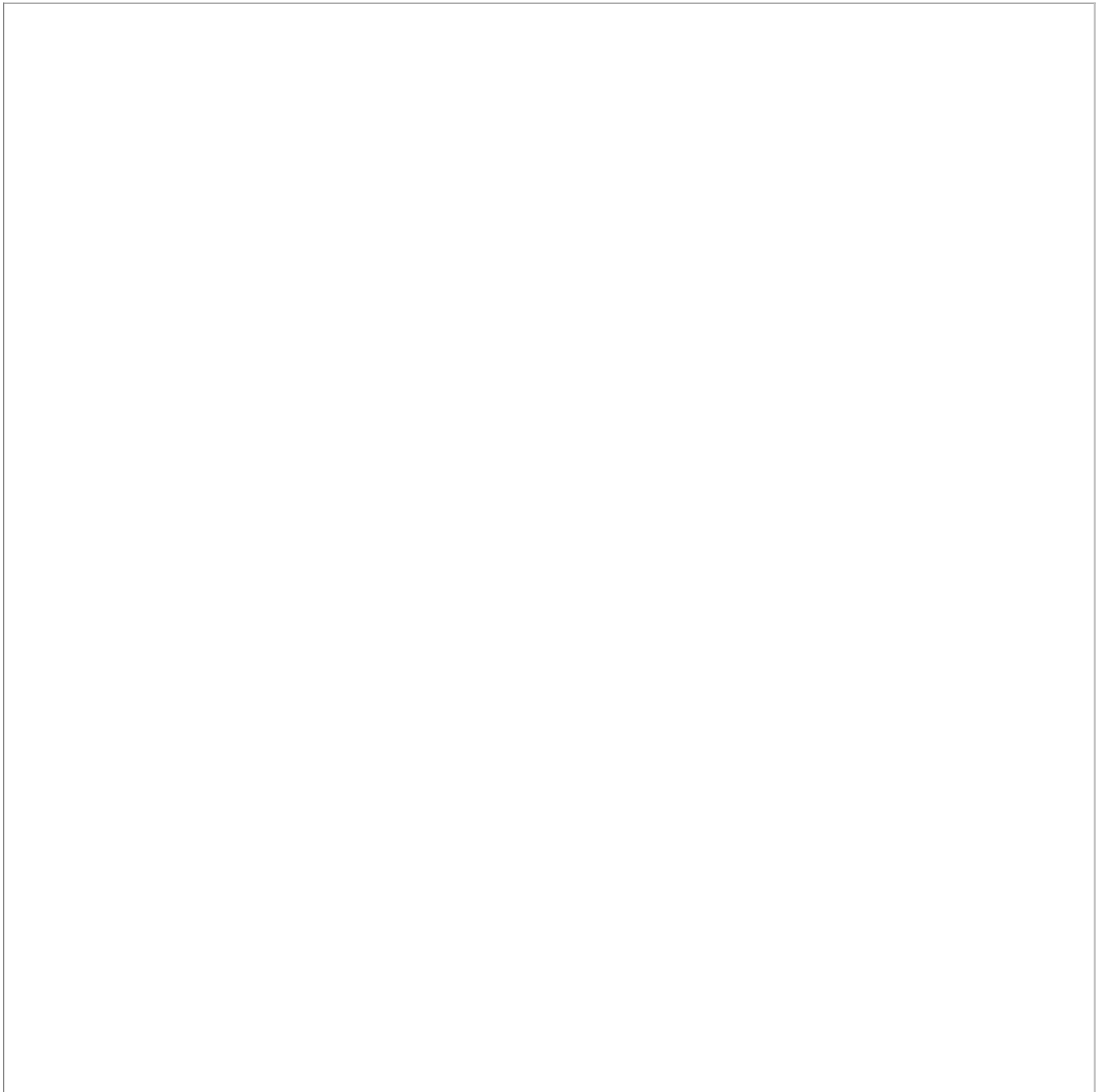
When you need the data on your local device, you can connect directly when the Pozyx software is running (as can be seen in the tray application). Connecting locally is recommended for real-time applications as it provides a very low latency.

A lot of programming languages provide MQTT libraries. The following Python snippet subscribes to the local MQTT stream using paho-mqtt:

```
1  import paho.mqtt.client as mqtt
2  import ssl
3
4  host = "localhost"
5  port = 1883
6  topic = "tags"
7
8  def on_connect(client, userdata, flags, rc):
9      print(mqtt.connack_string(rc))
10
11  # callback triggered by a new Pozyx data packet
12  def on_message(client, userdata, msg):
13      print("Positioning update:", msg.payload.decode())
14
15  def on_subscribe(client, userdata, mid, granted_qos):
16      print("Subscribed to topic!")
17
18  client = mqtt.Client()
19
20  # set callbacks
21  client.on_connect = on_connect
22  client.on_message = on_message
23  client.on_subscribe = on_subscribe
24  client.connect(host, port=port)
25  client.subscribe(topic)
26
27  # works blocking, other, non-blocking, clients are available too.
28  client.loop_forever()
```

1.6.3 Connect through the cloud stream

By connecting through the cloud, you can receive your data anywhere, without requiring to be on the same device or the same network. The Pozyx cloud offers the MQTT data in a secure way. To gain access to the data of your setup, you must provide the correct credentials. These can be found in the application by going to Settings > API Keys. Here you can generate a secure key. Once you have created your key, you are ready to connect. By deleting a key, a consumer using that key will no longer be able to access to the data.



The MQTT service, provided as a WebSocket, is hosted on:

```
host = mqtt.cloud.pozyxlabs.com7  
port = 443
```

The remaining connection setting - topic, username, and password - can be found in the application by going to Settings > API Keys.


The following Python snippet subscribes to the cloud MQTT stream, using the credentials obtained as described above:

⁷ <http://mqtt.cloud.pozyxlabs.com>


```

1  import paho.mqtt.client as mqtt
2  import ssl
3
4  host = "mqtt.cloud.pozyxlabs.com"
5  port = 443
6  topic = "" # your mqtt topic
7  username = "" # your mqtt username
8  password = "" # your generated api key
9
10 def on_connect(client, userdata, flags, rc):
11     print(mqtt.connack_string(rc))
12
13 # Callback triggered by a new Pozyx data packet
14 def on_message(client, userdata, msg):
15     print("Positioning update:", msg.payload.decode())
16
17 def on_subscribe(client, userdata, mid, granted_qos):
18     print("Subscribed to topic!")
19
20 client = mqtt.Client(transport="websockets")
21 client.username_pw_set(username, password=password)
22
23 # sets the secure context, enabling the WSS protocol
24 client.tls_set_context(context=ssl.create_default_context())
25
26 # set callbacks
27 client.on_connect = on_connect
28 client.on_message = on_message
29 client.on_subscribe = on_subscribe
30 client.connect(host, port=port)
31 client.subscribe(topic)
32
33 # works blocking, other, non-blocking, clients are available too.
34 client.loop_forever()

```

 Please note that the cloud MQTT service caps the maximum update rate. This value can be set under settings. By default the update rate is capped to 10 Hz.

The MQTT data is sent as a JSON array. A single-packet-array with all sensor data enabled will look like this:

```
1  [{
2    {
3      "version": "1",
4      "tagId": "24576",
5      "success": true,
6      "timestamp": 1524496105.895,
7      "data": {
8        "tagData": {
9          "gyro": {
10             "x": 0,
11             "y": 0,
12             "z": 0
13          },
14          "magnetic": {
15             "x": 0,
16             "y": 0,
17             "z": 0
18          },
19          "quaternion": {
20             "x": 0,
21             "y": 0,
22             "z": 0,
23             "w": 0
24          },
25          "linearAcceleration": {
26             "x": 0,
27             "y": 0,
28             "z": 0
29          },
30          "pressure": 0,
31          "maxLinearAcceleration": 0
32        },
33        "anchorData": [],
34        "coordinates": {
35          "x": 1000,
36          "y": 1000,
37          "z": 0
38        },
39        "acceleration": {
40          "x": 0,
41          "y": 0,
42          "z": 0
43        },
44        "orientation": {
45          "yaw": 0,
46          "roll": 0,
47          "pitch": 0
48        },
49        "metrics": {
50          "latency": 2.1,
51          "rates": {
52            "update": 52.89,
53            "success": 52.89
54          }
55        }
56      }
57    }
58  ]
59 }
```

```
56     }  
57     }  
58   }]
```

2 Arduino

Connect to the developer tag using an Arduino. Download the Arduino Library and get started!

2.1 Downloads

- The **Arduino IDE**. Download the IDE from the [Arduino website](https://www.arduino.cc/en/Main/Software)⁸ and install it.
- The **Pozyx Arduino Library**. The library can be obtained from [Github](https://github.com/pozyxLabs/Pozyx-Arduino-library)⁹ or [downloaded from the zip file directly](https://github.com/pozyxLabs/Pozyx-Arduino-library/archive/master.zip)¹⁰. Once downloaded, open the Arduino IDE and go to Sketch > Include Library > Add .ZIP library... and select the downloaded zip file. The examples should now show up under File > Examples > Pozyx (a restart of the program may be necessary first).
- The **Pozyx Arduino Library documentation** can be found here: <https://ardupozyx.readthedocs.io/>
- [Tutorial 1: Ready to range \(Arduino\)](#)(see page 44)
- [Tutorial 2: Ready to localize \(Arduino\)](#)(see page 50)
- [Tutorial 3: Orientation 3D \(Arduino\)](#)(see page 60)
- [Tutorial 4: Multitag positioning \(Arduino\)](#)(see page 65)
- [Chat room \(Arduino\)](#)(see page 67)
- [Configuration of the UWB parameters \(Arduino\)](#)(see page 70)
- [Changing the network ID \(Arduino\)](#)(see page 72)
- [Combination of cloud and Arduino](#)(see page 73)
- [Troubleshoot basics \(Arduino\)](#)(see page 76)

2.2 Tutorial 1: Ready to range (Arduino)

2.2.1 Ready to range

This is the first general Pozyx tutorial for the Creator system. If you haven't read the getting started, please do so and install the necessary tools and libraries.

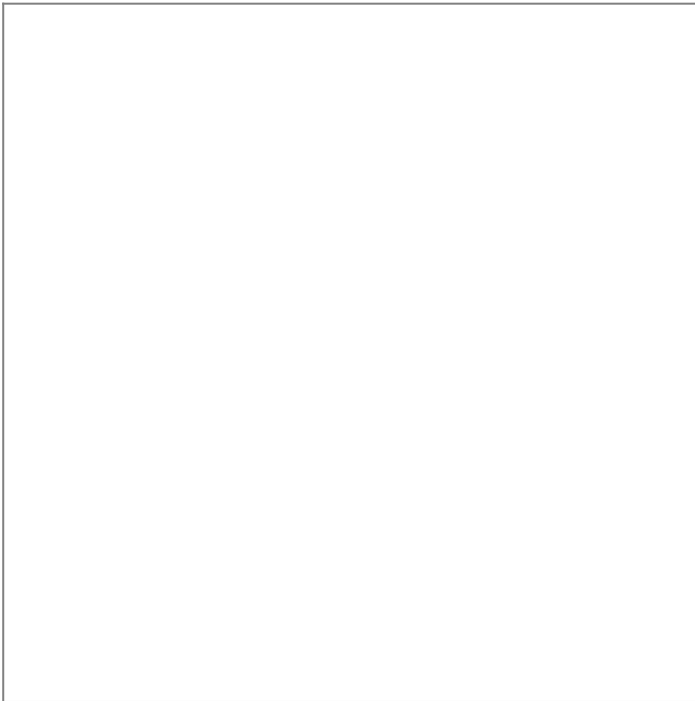
This example requires two Pozyx devices and one supported Arduino device. Remember that we recommend the Arduino Uno Rev. 3. If all tools are installed, open up the ready to range example in the Arduino IDE under File > Examples > Pozyx > ready_to_range.

In this example, the distance between the two devices is measured and the onboard LEDs will show in which range the devices are to each other. The destination Pozyx's LEDs will be doing the same, commanded by our local Pozyx. The LEDs' changes with respect to the distance is sketched in the figure below:

⁸ <https://www.arduino.cc/en/Main/Software>

⁹ <https://github.com/pozyxLabs/Pozyx-Arduino-library>

¹⁰ <https://github.com/pozyxLabs/Pozyx-Arduino-library/archive/master.zip>



2.2.2 Plug and play

To see the tutorial in action before we delve into the code, the parameters will need to be changed to match your destination device's ID.

You also have to set the Arduino IDE's serial monitor's baudrate to 115200. At the top of the Arduino sketch, we find the sketch's parameters:

```
uint16_t destination_id = 0x6670; // the network ID of the ranging destination
signed int range_step_mm = 1000; // distance between each LED lighting up.
uint8_t ranging_protocol = POZYX_RANGE_PROTOCOL_PRECISION; // the ranging protocol to use.
uint16_t remote_id = 0x605D;      // the network ID of the remote device
bool remote = false;              // whether to use the given remote device for ranging
```

You'll have to change the `destination_id` parameter to the ID of your destination Pozyx. Optionally, you can change the `range_step_mm` as well, which will make the distance range indicated by the LEDs either higher or lower depending on your change. `remote_id` and `remote` will allow measuring the distance between two remote Pozyx devices, and will be explained at the end of this tutorial, so don't change those for now.

The `ranging_protocol` is a new feature that allows changing between two ranging protocols: `POZYX_RANGE_PROTOCOL_FAST` and `POZYX_RANGE_PROTOCOL_PRECISION`. Precision is slower, but more precise and can be used on longer ranges. The fast protocol needs to warm up for about 100ms before. If this has piqued your interest, definitely check the system performance section in the datasheet!

If you've correctly changed your device ID, run the example! You should see that the LEDs of your two Pozyx devices now burn identically, and change when you move them closer or further apart. The output you're receiving will look like this:

```
16691ms, 7564mm, -93dBm
16753ms, 7718mm, -91dBm
```

16830ms, 7779mm, -92dBm
etc...

The first value is the device's timestamp of the range measurement. The second value then is the measured distance between both devices. The third value, expressed in dBm, signifies the signal strength, varying between -79 and -103 dB typically. You might be reading either:

0ms, 0mm, 0dB

or

ERROR: ranging

If this is the case, you probably miswrote the ID (did you write it with the 0x prefix?) or your device is not in range currently. As the LEDs will only indicate differences of a meter by default, keeping the distance between a couple of meters for this first try is recommended. If the error persists despite having put in everything correctly, check out the troubleshooting guide.

If you're getting gibberish, you probably forgot to set your serial monitor's baudrate to 115200.

2.2.3 The code explained

Now that you've had a taste of what exactly this example provides, let's look at the code that provides this functionality. We'll begin with the imports and the setup functions, and then go into the continuous loop.

2.2.3.1 Imports

```
#include <Pozyx.h>
#include <Pozyx_definitions.h>
#include <Wire.h>
```

We can see that the sketch requires both "Pozyx.h" and "Pozyx_definitions.h", and the "Wire.h" file for the I2C.. "Pozyx.h" provides the library's functionality, "while Pozyx_definitions.h" provides constants such as POZYX_SUCCESS and POZYX_FAILURE, register addresses... these are useful to keep your code working with both current, previous, and future versions of the library.

2.2.3.2 Setup

This function will run only once, when the Arduino is powered on or reset. It sets up the serial communication at 115200 baud, so be sure to set your Serial Monitor to this same baudrate if you hadn't already. The Pozyx is then initialized by calling Pozyx.begin() , which checks whether the Pozyx is connected and working properly.

```

1 void setup(){
2   Serial.begin(115200);
3   if(Pozyx.begin() == POZYX_FAILURE){
4     Serial.println("ERROR: Unable to connect to POZYX shield");
5     Serial.println("Reset required");
6     delay(100);
7     abort();
8   }
9   // setting the remote_id back to NULL will use the local Pozyx
10  if (!remote){
11    remote_id = NULL;
12  }
13  Serial.println("-----POZYX RANGING V1.0-----");
14  Serial.println("NOTES:");
15  Serial.println("- Change the parameters:");
16  Serial.println("\tdestination_id (target device)");
17  Serial.println("\trange_step (mm)");
18  Serial.println();
19  Serial.println("- Approach target device to see range and");
20  Serial.println("led control");
21  Serial.println("-----POZYX RANGING V1.0-----");
22  Serial.println();
23  Serial.println("START Ranging:");
24  // make sure the pozyx system has no control over the LEDs, we're the boss
25  uint8_t led_config = 0x0;
26  Pozyx.setLedConfig(led_config);
27  // do the same with the remote device
28  Pozyx.setLedConfig(led_config, destination_id);
29 }

```

Depending on the value of the remote parameters, we set the `remote_id` to use either the local or a remote Pozyx. After initializing the Pozyx, we configure both the local and destination Pozyx's LEDs to be in our control instead of displaying system status, which is their default behavior. This is done by setting the `POZYX_CONFIG_LEDS` register's value to `0b00000000` on both devices. While the library's functionality hides away a lot of low-level functionality, it's important to know how to access registers directly low-level and to find your way in the register overview for your own personal goals. These core low-level functions are:

```

static int regRead(uint8_t reg_address, uint8_t *pData, int size);
static int regWrite(uint8_t reg_address, const uint8_t *pData, int size);
static int regFunction(uint8_t reg_address, uint8_t *params, int param_size, uint8_t *pData, int size);

```

and the remote equivalents

```

static int remoteRegWrite(uint16_t destination, uint8_t reg_address, uint8_t *pData, int size);
static int remoteRegRead(uint16_t destination, uint8_t reg_address, uint8_t *pData, int size);
static int remoteRegFunction(uint16_t destination, uint8_t reg_address, uint8_t *params, int param_size,
uint8_t *pData, int size);

```

2.2.3.3 Loop

Let's move on to the loop function, which will run continuously and executes the ranging functionality.

```

1 void loop(){
2     device_range_t range;
3     int status = 0;
4     // let's perform ranging with the destination
5     if(!remote)
6         status = Pozyx.doRanging(destination_id, &range);
7     else
8         status = Pozyx.doRemoteRanging(remote_id, destination_id, &range);
9     if (status == POZYX_SUCCESS){
10        Serial.print(range.timestamp);
11        Serial.print("ms, ");
12        Serial.print(range.distance);
13        Serial.print("mm, ");
14        Serial.print(range.RSS);
15        Serial.println("dB");
16        // now control some LEDs; the closer the two devices are, the more LEDs will be lit
17        if (ledControl(range.distance) == POZYX_FAILURE){
18            Serial.println("ERROR: setting (remote) leds");
19        }
20    }
21    else{
22        Serial.println("ERROR: ranging");
23    }
24 }
```

In the loop function, ranging is performed with the Pozyx library's `doRanging` function. The range information will be contained in `device_range`.

When we want to perform ranging on a remote Pozyx, we use `doRemoteRanging` instead. The `device_range` variable is an instance of the `device_range_t` structure. Looking at its definition (in `Pozyx.h`) we can see that the device range consists of three variables providing information about the measurement. By using the `__attribute__((packed))` attribute, the struct's bytes are packed together.

```

1 typedef struct __attribute__((packed))_device_range {
2     uint32_t timestamp;
3     uint32_t distance;
4     int8_t RSS;
5 }device_range_t;
```

In a wireless system, we don't want to waste transmission time on unused padding bytes, after all. The library provides similar structures for all relevant data structures related to Pozyx, which you'll encounter if you follow the other tutorials and get into further reading. The measurement consists of the timestamp in milliseconds, the measured distance in mm and the RSS-value (the received signal strength) expressed in dBm. Depending on the UWB settings, -103 dBm is more or less the lowest RSS at which reception is possible, and -80 is a normal value for devices close to each other. Upon a successful measurement, these three variables are printed and the LEDs are updated with the `ledControl` function.

2.2.3.4 ledControl

```

1  int ledControl(uint32_t range){
2      int status = POZYX_SUCCESS;
3      // set the LEDs of the pozyx device
4      status &= Pozyx.setLed(4, (range < range_step_mm), remote_id);
5      status &= Pozyx.setLed(3, (range < 2*range_step_mm), remote_id);
6      status &= Pozyx.setLed(2, (range < 3*range_step_mm), remote_id);
7      status &= Pozyx.setLed(1, (range < 4*range_step_mm), remote_id);
8      // set the LEDs of the destination pozyx device
9      status &= Pozyx.setLed(4, (range < range_step_mm), destination_id);
10     status &= Pozyx.setLed(3, (range < 2*range_step_mm), destination_id);
11     status &= Pozyx.setLed(2, (range < 3*range_step_mm), destination_id);
12     status &= Pozyx.setLed(1, (range < 4*range_step_mm), destination_id);
13     // status will be zero if setting the LEDs failed somewhere along the way
14     return status;
15 }

```

The ledControl function turns the LEDs on or off if they fall within a certain range, using a simple boolean expression, for example, `range < 3*range_step_mm`, which returns true if the distance is within three times the `range_step_mm` parameter.

Nothing is stopping you from designing your own ledControl function, where you can use other indexes or, as a challenge, even use the four LEDs as 4 bits indicating up to sixteen `range_step_mm` differences instead of four.

2.2.4 Remote operation

As you have probably seen when looking at the code and the register functions, Pozyx is capable of communicating with remote Pozyx devices: thanks to UWB communication, we can perform every local functionality on a remote device as well. In this tutorial, this can be done by changing the `remote_id` and remote parameters. Setting `remote = true`; will perform all the functionality on a device with ID `remote_id`, so you will need to set that ID correctly as well and own three devices to perform remote ranging.

A general rule with the Pozyx API is that almost every function has `remote_id` as a keyword argument, which allows the writing of code that can easily switch between being executed locally or remotely, as seen in this example. When `remote_id` is uninitialized, the function will be executed locally.

The disadvantages of working remotely need to be kept in mind, however. The possibility of losing connection, something that is much harder with a cable. A delay, as the UWB communication between both devices is not instantaneous. And you need a local controller, which is an extra device in your setup. In turn, you gain a lot of mobility, as the remote device only needs to be powered, not needing a controller.

This concludes the first Pozyx example. In the next example, we'll cover positioning and the necessary setup to perform positioning successfully.

2.3 Tutorial 2: Ready to localize (Arduino)

2.3.1 Ready to localize

This is the second Pozyx tutorial, in which we'll go through the process of performing (remote) positioning with the Pozyx system. If you missed the first one, check that one out first, as each tutorial assumes knowledge of the concepts explained in the ones before.

For this example, you need to own at least the contents of the Ready to Localize kit and a supported Arduino device. Remember that we recommend the Arduino Uno Rev. 3. If all tools are installed, open up the ready to localize example in the Arduino IDE under File > Examples > Pozyx > ready_to_localize.

In this example, we will first set up and measure the anchor locations to perform positioning, after which we'll be able to get accurate position data of the tag, relative to the anchor setup. We will go in detail through other Pozyx core concepts as well, such as how to check the Pozyx' device list, and how to read its error status. Mastering these debugging tools now will make using Pozyx in your own projects easier

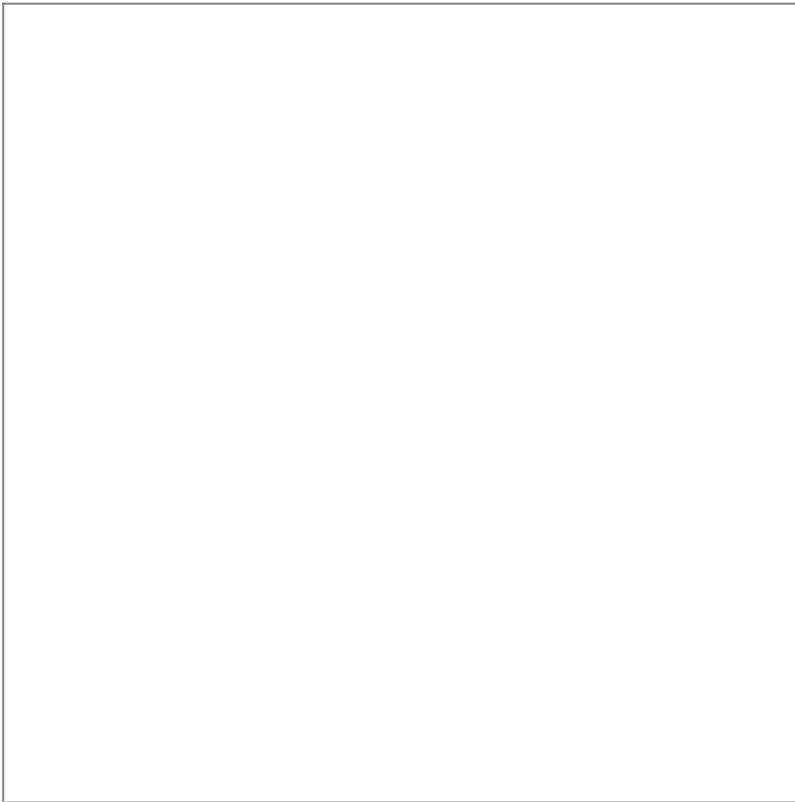
2.3.2 Anchor setup and measurement

2.3.2.1 Anchor setup

The Pozyx positioning system requires that the four anchors are placed inside the area where you wish to do positioning. In the guide 'Where to place the anchors?', it is explained how to place the anchors for the best possible positioning accuracy. The rules of thumb for the anchor placement were:

- Place the anchors high and in line-of-sight of the user.
- Spread the anchors around the user, never all on a straight line!
- Place the anchors vertically with the antenna at the top, power plug at the bottom.
- For 3D positioning: place the anchors at different heights.

It's also important to keep metallic objects out of immediate range of the antennas.



Before you install the anchors (with the provided Velcros or screws) on the walls or ceiling, it is usually a good idea to make a small sketch of the room, writing down the anchors' IDs and sketching where they will be placed. You can find the network ID as the hexadecimal number on the label adorning your anchor.

Remember that, for optimal antenna performance, it is recommended to place the anchors vertically with their antenna at the top, and to orient your tags that will be positioning vertically as well. Also make sure that no heavy metal objects are placed near the antenna, as this might degrade performance. We stress this again because it's just that important.

2.3.2.2 Measurement

While there are different ways to measure your distance, if you are serious about using Pozyx we recommend to have a laser measurer. This will be much more convenient, and much more accurate, than trying to measure out several meters between your anchors with a conventional foldable or extendable We recommend placing them in an approximate rectangle as in the image above. This allows you to simplify both measurement and setup, and to follow along better with the example.

2.3.2.3 The Pozyx coordinate system

Pozyx determines its position relative to the anchor coordinates you supplied it with. This gives you the freedom to position and rotate your axis in any way you want. This is shown in the images below:



You can see that the coordinate system on the left, which is conventional to the room's shape and orientation, is made by using anchor 0x256B as an element of the x-axis, giving it a zero y-coordinate. Then the y-axis is perpendicular to this, and anchor 0x3325 therefore has a non-zero x-component to fit into this orthogonal system. Anchor 0x1156 is selected as the origin, again for convenience. This origin doesn't need to be one of the anchors, however, and you could use the center of the room as the origin point just as well, modifying the anchor's coordinates accordingly: anchors 0x1156 and 0x3325 would have a negative x-component, while 0x2568 and 0x4244 have a positive coordinate. In turn, 0x1156 and 0x256B have a negative y-component while 0x3325 and 0x4244 will have a positive one.

In the right example, the origin is chosen to be anchor 0x3325's location, and the x-axis is matched with the path between anchors 0x3325 and 0x1156.

Let's go over the plan of attack in measuring out your setup, following the approach shown on the left:

- Use the antenna's center as Pozyx's position in space as best as you can.
- Pick one of your anchors as the origin, like we did 0x1156.
- Pick the anchor on the same wall to be the one defining your x-axis. We picked 0x2568.
- Measure out the horizontal distance between both anchors. Not a direct distance as this will include the vertical distance as well (Pythagoras). This will be that anchor's x-coordinate. In our case, this was 4.5 m.
- Now measure the distance to the opposite wall. Mark this point on the wall, as this is the x-axis's zero value on that wall. If both walls are parallel, and there are anchors attached directly to this wall, you can set their y-coordinate to this distance. In our case, we could set 0x4244's y-coordinate directly to 3.5 m.
- Now measure the x-coordinates of every anchor on that wall directly using the point you marked earlier, again assuming both walls are parallel. Measurements can be complicated if they are not, so use more reference points then.
- If there are anchors a bit apart from the wall, like 0x3325, be sure to account for this in its y-coordinate.

In this example, we've used the approach described above. We'll also assume the anchors are on heights between 1.1 and 2 meter, leading to these coordinates for each anchor:

```
0x1156: (0, 0, 1500)
0x256B: (4500, 0, 1800)
0x3325: (500, 3300, 1100)
0x4244: (4450, 3500, 2000)
```

These are the coordinates used in the example, but you can't copy these! You will have to change both anchor IDs and coordinates to match your own setup. All these coordinates are expressed in millimeters.

2.3.3 Plug and play

To get the example working and see what exactly is happening before we delve into the code, we'll need to change the parameters to match your device IDs and the coordinates you measured.

Don't forget to set the Arduino IDE's serial monitor's baudrate to 115200. At the top of the Arduino sketch, we find the sketch's parameters. You can see that there's an array for the anchor IDs and each of the anchor coordinates. You'll have to translate your measurement to these parameters, where each anchor's properties has the same index in its respective array. We translated the example setup from above, as a reference.

```

1  uint16_t remote_id = 0x6000;
2  bool remote = false;                                // set this to true to use the remote
   ID
3  boolean use_processing = false;                      // set this to true to output data for
   the processing sketch
4  uint8_t num_anchors = 4;                             // the number of anchors
5  uint16_t anchors[4] = {0x1156, 0x256B, 0x3325, 0x4244}; // the network id of the anchors:
   change these to the network ids of your anchors.
6  int32_t anchors_x[4] = {0, 4500, 500, 4450};        // anchor x-coorindates in mm
7  int32_t anchors_y[4] = {0, 0, 3300, 3500};          // anchor y-coordinates in mm
8  int32_t heights[4] = {1500, 1800, 1100, 2000};      // anchor z-coordinates in mm
9  uint8_t algorithm = POZYX_POS_ALG_UWB_ONLY;
10 uint8_t dimension = POZYX_3D;
11 int32_t height = 1000;
```

You will also see the `remote_id`, `remote`, `algorithm`, `dimension`, and `use_processing` parameters. `remote_id` and `remote` should be familiar from the first tutorial. `use_processing` will be used for the visualization. `algorithm`, `dimension`, and `height` will allow for customization regarding the positioning, and we'll get back to these when

we're looking at the code in detail. Leave these parameters unchanged for now, which will result in 3D positioning being done with the UWB-only algorithm.

Now that you've done all the plugging, it's time for play. Run the example, and if all goes well, you're now looking at coordinates filling up the window, after the manual anchor configuration is checked and printed. As you are using your local device, the ID will be set to 0.

```
POS ID 0x0000, x(mm): 1231 y(mm): 2354 z(mm): 1167
POS ID 0x0000, x(mm): 1236 y(mm): 2241 z(mm): 1150
etc...
```

That's that! You're now getting accurate position coordinates from the Pozyx! You might also be seeing one of the following:

```
POS ID 0x0000, x(mm): 0 y(mm): 0 z(mm): 0
or
ERROR configuration on ID 0x0000, error code 0xXX
or
ERROR positioning on ID 0x0000, error code 0xXX
or
Coordinates that are nothing even remotely close to what they should be
```

If so, you probably miswrote one of the anchor IDs or entered either wrong or mixed up coordinates, causing the positioning algorithm to fail. If the error persists despite having put in everything correctly, check out the troubleshooting guide. Now that you've seen the positioning in action, let's look at the code that made this possible.

2.3.4 The code explained

We will now cover the essential code to get positioning working, but there's a lot more code in the file. The extras segment will cover how to access the Pozyx's device list, how to retrieve error codes returned by the Pozyx, and how to retrieve additional sensor data each time you position.

2.3.4.1 Setup and manual calibration

Both the serial communication and Pozyx are initialized in the setup function.

```

1 void setup(){
2   Serial.begin(115200);
3   if(Pozyx.begin() == POZYX_FAILURE){
4     Serial.println(F("ERROR: Unable to connect to POZYX shield"));
5     Serial.println(F("Reset required"));
6     delay(100);
7     abort();
8   }
9   if(!remote){
10    remote_id = NULL;
11  }
12  Serial.println(F("-----POZYX POSITIONING V1.0-----"));
13  Serial.println(F("NOTES:"));
14  Serial.println(F("- No parameters required.));
15  Serial.println();
16  Serial.println(F("- System will auto start anchor configuration"));
17  Serial.println();
18  Serial.println(F("- System will auto start positioning"));
19  Serial.println(F("-----POZYX POSITIONING V1.0-----"));
20  Serial.println();
21  Serial.println(F("Performing manual anchor configuration:"));
22  // clear all previous devices in the device list
23  Pozyx.clearDevices(remote_id);
24  // sets the anchor manually
25  setAnchorsManual();
26  printCalibrationResult();
27  delay(2000);
28  Serial.println(F("Starting positioning: "));
29 }

```

The setup function is straightforward. After its initialization of the Pozyx, it performs the anchor configuration. It first clears the Pozyx's device list using `clearDevices()`, and then manually adds the anchors in `setAnchorsManual()` we set up and measured out. Let's look at how this is done:

```

1 void setAnchorsManual(){
2   for(int i = 0; i < num_anchors; i++){
3     device_coordinates_t anchor;
4     anchor.network_id = anchors[i];
5     anchor.flag = 0x1;
6     anchor.pos.x = anchors_x[i];
7     anchor.pos.y = anchors_y[i];
8     anchor.pos.z = heights[i];
9     Pozyx.addDevice(anchor, remote_id);
10  }
11  if (num_anchors > 4){
12    Pozyx.setSelectionOfAnchors(POZYX_ANCHOR_SEL_AUTO, num_anchors);
13  }
14 }

```

We define each anchor as a `device_coordinates_t` object, which takes an ID, a flag to indicate what kind of device it is – `POZYX_ANCHOR` or `POZYX_TAG` – and the device's coordinates. We then add this anchor to the device's stored

device list, which it will use when positioning. If you'd use more than four anchors, the device's anchor selection is set to automatically use all available anchors from this set.

2.3.4.2 Loop

Now that we've properly configured the device's device list with our set of anchors, let's look at just how easy the actual positioning is in the short `loop()` function:

```

1  void loop(){
2      coordinates_t position;
3      int status;
4      if(remote){
5          status = Pozyx.doRemotePositioning(remote_id, &position, dimension, height, algorithm);
6      }else{
7          status = Pozyx.doPositioning(&position, dimension, height, algorithm);
8      }
9      if (status == POZYX_SUCCESS){
10         // prints out the result
11         printCoordinates(position);
12     }else{
13         // prints out the error code
14         printErrorCode("positioning");
15     }
16 }
```

We first create a new object that will contain the Pozyx's measured coordinates, after which we call the Pozyx's `doPositioning` function, which will perform the positioning algorithm and store the coordinates, measured respectively to the anchors, in the position object. That's essentially the entire positioning loop! If `doPositioning` returns `POZYX_SUCCESS`, the position will be printed in a human readable way. We see that we can pass the positioning's algorithm, dimension, and Pozyx height (used in 2.5D) as parameters in the positioning. There are three dimensions supported by Pozyx: `POZYX_2D`, `POZYX_2_5D`, and `POZYX_3D`.

In 2D the anchors and tags must all be located in the same horizontal plane. This is not the case for semi-3D or 3D positioning. In semi-3D positioning the height of the tag must be supplied in the height parameter (for example, when the tag is mounted on a driving robot with fixed height). The reason why semi-3D exists is because in many cases it is not possible to obtain an accurate estimate for the z-coordinate in 3D-positioning. This is a result of how the anchors are placed and is explained in the guide 'Where to place the anchors?'. As a final parameter you can supply which algorithm to use. Possible values are `POZYX_POS_ALG_UWB_ONLY` and `POZYX_POS_ALG_TRACKING`. By default `POZYX_POS_ALG_UWB_ONLY` is used. For more information about the algorithms we refer to the register documentation for `POZYX_POS_ALG`. We've defined the algorithm as a parameter, so you can change the algorithm in the paramaters section instead of directly doing so in the function.

Some example usages:

```
status = Pozyx.doPositioning(&position, POZYX_2_5D, 1000);
```

semi-3D positioning with the height of the tag fixed at 1000mm (about 3.3feet).

```
status = Pozyx.doPositioning(&position, POZYX_3D);
```


3D positioning, this requires at least 4 anchors. Note that the anchors must be placed at different heights to obtain a good accuracy of the z-coordinate.

2.3.5 Remote positioning

Positioning the Pozyx attached to your PC means that, except for when you have a long cable, you'll also need to be able to move your PC around if you want to position over a larger area than your desk. This would also mean that if you'd want to track objects, you'd need to attach a processing unit to these objects as well, instead of only the Pozyx. Luckily, the Pozyx attached to your computer can act as a master device, commanding one or more remote 'slaves'.

To position a remote device, you need to do the same as on a local device: put anchors in its device list that it will use for the positioning. A common misconception about the use of Pozyx is that configuring the anchors on your local device will make remote devices capable of positioning straight off the bat, but this isn't the case. You will notice that the `addDevice` function in `setAnchorsManual` adds the device to a remote device if remote positioning is enabled, thusly configuring the anchors on the remote device and not on the local one.

2.3.6 Extras

While not necessary for positioning, the added functionality in the code can go a long way for extending the use of Pozyx or when things go wrong, so going over these extras is recommended if you want to take things further without needing to figure things out yourself.

2.3.6.1 Printing the configuration result

To find out whether the calibration was in fact successful, we will retrieve the Pozyx's device list. This is the reverse of the configuration step, as we now retrieve the IDs and coordinates in turn from the Pozyx. Pozyx requires this to be done in several steps:

- Firstly we retrieve the size of the device list through `getDeviceListSize`.
- We use this size to create an appropriately sized device list container
- We retrieve the IDs in the device's device list using this container with `getDeviceIds`. You can also use `getTagIds` and `getAnchorIds` to have more control over which device IDs you're getting.
- Now we can get the device coordinates belonging to each of these IDs using `getDeviceCoordinates`

This is what is done in the code below, and we print the anchor's ID with its retrieved coordinates. If these don't match the anchors you passed to the device, something went wrong and it is recommended to try again. If this keeps failing, try going through the troubleshooting.

```

1  void printCalibrationResult(){
2      uint8_t list_size;
3      int status;
4      status = Pozyx.getDeviceListSize(&list_size, remote_id);
5      Serial.print("list size: ");
6      Serial.println(status*list_size);
7      if(list_size == 0){
8          printErrorCode("configuration");
9          return;
10     }
11     uint16_t device_ids[list_size];
12     status &= Pozyx.getDeviceIds(device_ids, list_size, remote_id);
13     Serial.println(F("Calibration result:"));
14     Serial.print(F("Anchors found: "));
15     Serial.println(list_size);
16     coordinates_t anchor_coor;
17     for(int i = 0; i < list_size; i++)
18     {
19         Serial.print("ANCHOR,");
20         Serial.print("0x");
21         Serial.print(device_ids[i], HEX);
22         Serial.print(",");
23         Pozyx.getDeviceCoordinates(device_ids[i], &anchor_coor, remote_id);
24         Serial.print(anchor_coor.x);
25         Serial.print(",");
26         Serial.print(anchor_coor.y);
27         Serial.print(",");
28         Serial.println(anchor_coor.z);
29     }
30 }

```

2.3.6.2 Printing the error code

Pozyx's operation can misbehave due to various reasons. Other devices not being in range, being on different settings, or another firmware version... As you're getting started with Pozyx, it's hard to keep track of where exactly things go wrong, and it's for this reason that Pozyx keeps track of what went wrong in the error status register, POZYX_ERRORCODE. The error code can be read in two ways, one uses the `getErrorCode` function to directly read out the value from the error register, while the other, through `getSystemError`, returns a more verbose error message, representing the relative error textually. For example, as the error code would be 0x05, `getSystemError` would return "Error 0x05: Error reading from a register from the I2C bus". While the latter is ideal when working in a printed environment, if you work with visualizations it's less than ideal to handle this entire string and you can perform custom functionality with the error code.

```

1  void printErrorCode(String operation){
2      uint8_t error_code;
3      if (remote_id == NULL){
4          Pozyx.getErrorCode(&error_code);
5          Serial.print("ERROR ");
6          Serial.print(operation);
7          Serial.print(", local error code: 0x");
8          Serial.println(error_code, HEX);
9          return;
10     }
11     int status = Pozyx.getErrorCode(&error_code, remote_id);
12     if(status == POZYX_SUCCESS){
13         Serial.print("ERROR ");
14         Serial.print(operation);
15         Serial.print(" on ID 0x");
16         Serial.print(remote_id, HEX);
17         Serial.print(", error code: 0x");
18         Serial.println(error_code, HEX);
19     }else{
20         Pozyx.getErrorCode(&error_code);
21         Serial.print("ERROR ");
22         Serial.print(operation);
23         Serial.print(", couldn't retrieve remote error code, local error: 0x");
24         Serial.println(error_code, HEX);
25     }
26 }

```

In this example, simple but comprehensive error checking is executed. If the positioning is purely local, the local error is read. When remote positioning fails, indicated by the positioning function returning `POZYX_FAILURE`, the error register is read. If the error couldn't be read remotely, the error is read locally. We are using `getErrorCode` instead of `getSystemError` because this allows us to efficiently send the error data to the visualization, and customize our error output. You can find the resulting error codes' meaning at the register documentation of `POZYX_ERRORCODE`.

2.3.6.3 Adding sensor information

Sensor information, such as orientation or acceleration, can easily be added to the code, as to return this sensor data every positioning loop. Adding orientation and/or acceleration allows you to get a better insight in the object you're tracking, but you'll have to account for the reduced positioning update rate caused by this additional operation. Especially remotely, this will delay your update rate. In this example code, not present in the actual script, we'll retrieve both orientation and acceleration.

```

1  void printOrientationAcceleration(){
2      orientation = euler_angles_t;
3      acceleration = acceleration_t;
4      Pozyx.getEulerAngles_deg(&orientation, remote_id);
5      Pozyx.getAcceleration_mg(&acceleration, remote_id);
6      Serial.print("Orientation: Heading:");
7      Serial.print(orientation.heading);
8      Serial.print(", Roll:");
9      Serial.print(orientation.roll);
10     Serial.print(", Pitch:");
11     Serial.print(orientation.pitch);
12     Serial.print(", acceleration: X:");
13     Serial.print(acceleration.x);
14     Serial.print(", Y:");
15     Serial.print(acceleration.y);
16     Serial.print(", Z:");
17     Serial.print(acceleration.z);
18 }

```

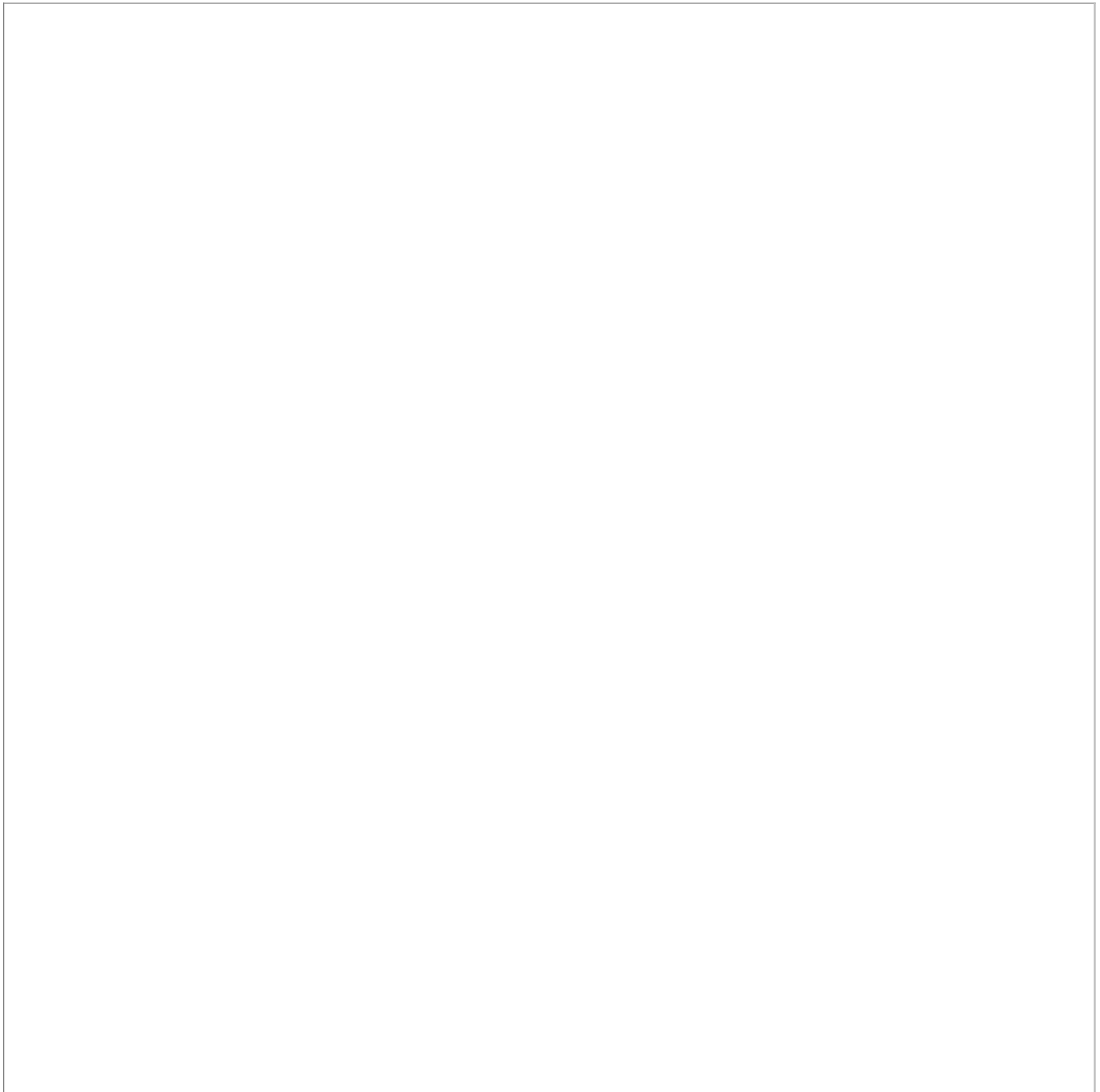
2.4 Tutorial 3: Orientation 3D (Arduino)

2.4.1 Downloads

In this example, the sensor data is read from the Pozyx developer tag (either the tag mounted on the Arduino, or a remote tag). To visualize all the data, we use Processing. Below a screenshot of the Processing program showing all the sensor data graphically.

This example requires one or two Pozyx developer tags and one Arduino. Open up the orientation 3D example in the Arduino IDE under File > Examples > Pozyx > orientation_3D.

Open up the pozyx_orientation3D.pde sketch in Processing, as this example will be directly visualized.



2.4.2 Plug and play

In Processing, make sure serial is set to true, and that your serialPort matches the one used by your Arduino. Upload the Arduino sketch, and, when this is done, press the play button in Processing.

You should be able to physically rotate your Pozyx tag now and directly see it rotate in Processing as well. There are plots on the side and at the top, but what do they all mean?

2.4.3 Understanding the sensor data

- **Acceleration (g):** the acceleration is measured along 3 axes (i.e., in 3 directions) which is shown by three different colors on the plot. The acceleration is expressed in g's (from gravity). 1g is exactly the acceleration from earth's gravitation pull ($1g = 9.81\text{m/s}^2$). Because the acceleration from gravity is always present and equal to 1g it can be used to determine how the device is tilted or pitched. Try rotating the Pozyx device by 90 degrees (slowly) and you will see the the acceleration change from one axis to the other.
- **Magnetic field strength (μT):** the magnetic field strength is also measured along 3 axes and is expressed in μT . It can be used to measure the earth's magnetic field which varies between 30 and $60\mu\text{T}$ over the earth's surface. The measured magnetic field strength can be used to estimate the magnetic north (similar to a compass). However, magnetic fields exist wherever electric current flows or magnetic materials are present. Because of this, they will influence the sensor data and it will become impossible to determine the magnetic north exactly. Try holding a magnet or something iron close to the Pozyx device and you will see the magnetic field strength fluctuate.
- **Angular velocity (deg/s):** The angular velocity is measured by the gyroscope and measures how fast the device rotates around itself. By integrating the angular velocity it is possible to obtain the angles of the device. However, due to drift this method does not give accurate results for a prolonged time. What is drift you wonder? Well if the device is standing still, the angular velocity should be exactly equal to zero. This is not the case however, it will be slightly different and this error will accumulate over time when integrating to angles.
- **3D Orientation:** The 3D orientation is shown by the 3D model in the middle. The orientation is computed by combining the sensor data from all three sensors together. By combining all sensors it is possible to overcome the limitations of each sensor separately. The 3D orientation can be expressed in Euler angles: yaw, pitch, roll or in quaternions. Quaternions are a mathematical representation using 4 numbers. In many situations, quaternions are preferred because they do not have singularity problems like the Euler angles.
- **Gravity vector (g):** If the Pozyx device is standing still, the acceleration is exactly equal to the gravity. The gravity vector is shown by the black line and is always pointing down. Notice that even when moving the device (which introduces an additional acceleration) the gravity vector still points down. This is due to the fusion algorithm that can separate gravity from an arbitrary acceleration;
- **Linear acceleration in body coordinates (g):** The linear acceleration is the acceleration that remains after the gravity has been removed. When you hold the device horizontal, pointed forward, and shake it from left to right the circle will also move from left to right in the plot. However, if you rotate the device by 90 degrees and shake it again from left to right, the circle will now move in a different direction. This is because the linear acceleration is expressed in body coordinates, i.e., relative to the device. Note that all the above sensor data is expressed in body coordinates.
- **Linear acceleration in world coordinates (g):** Once the orientation of the device is known, it is possible to express the acceleration in world coordinates. By doing this, the rotation of the device no longer affects the linear acceleration in the plot.

2.4.4 The code explained

We'll now go over the code that's needed to retrieve all sensor data. Looking at the code's parameters, we can see that we can once again use a remote Pozyx tag for this. This means that you could for example attach a Pozyx tag to a ball, and watch the ball's spin directly on your screen as well with the Pozyx tag.

2.4.4.1 Imports and setup

The imports are the default Pozyx library imports and Wire.h for the I²C functionality encountered in the previous tutorials.

The Pozyx and serial connection are then initialized, and the last measured time is set.

```

1  #include <Pozyx.h>
2  #include <Pozyx_definitions.h>
3  #include <Wire.h>
4
5  void setup()
6  {
7      Serial.begin(115200);
8      if(Pozyx.begin(false, MODE_INTERRUPT, POZYX_INT_MASK_IMU) == POZYX_FAILURE){
9          Serial.println("ERROR: Unable to connect to POZYX shield");
10         Serial.println("Reset required");
11         delay(100);
12         abort();
13     }
14     if(!remote)
15         remote_id = NULL;
16     last_millis = millis();
17     delay(10);
18 }

```

2.4.4.2 Loop

The main loop deserves to be elaborated on. The Pozyx's IMU sensors trigger an interrupt flag when there is new data available, and in the code we wait for this flag to trigger explicitly.

When it does, or when we read the sensor data remotely, the code will read out all sensor data and it's calibration status from the (remote) Pozyx, and then print it serially as comma separated values, which will be converted to standard units and interpreted by Processing. This is done in `getAllRawSensorData`, where the sensor registers are read directly and their values are stored in a `sensor_raw_t` struct. There is also a `getAllSensorData` function available, where this raw sensor data gets converted to standard units and is stored in a `sensor_data_t` struct, but we choose the former because we prefer this conversion to be done in Processing, on a powerful CPU, offsetting the Arduino's processing and allowing a faster retrieval rate of this IMU data.

When it doesn't, it tries again.

```

1 void loop(){
2     sensor_raw_t sensor_raw;
3     uint8_t calibration_status = 0;
4     int dt;
5     int status;
6     if(remote){
7         status = Pozyx.getRawSensorData(&sensor_raw, remote_id);
8         status &= Pozyx.getCalibrationStatus(&calibration_status, remote_id);
9         if(status != POZYX_SUCCESS){
10             return;
11         }
12     }else{
13         if (Pozyx.waitForFlag(POZYX_INT_STATUS_IMU, 10)){
14             Pozyx.getRawSensorData(&sensor_raw);
15             Pozyx.getCalibrationStatus(&calibration_status);
16         }else{
17             uint8_t interrupt_status = 0;
18             Pozyx.getInterruptStatus(&interrupt_status);
19             return;
20         }
21     }
22     dt = millis() - last_millis;
23     last_millis += dt;
24     // print time difference between last measurement in ms, sensor data, and calibration data
25     Serial.print(dt, DEC);
26     Serial.print(",");
27     printRawSensorData(sensor_raw);
28     Serial.print(",");
29     // will be zeros for remote devices as unavailable remotely.
30     printCalibrationStatus(calibration_status);
31     Serial.println();
32 }

```

The calibration status gives information about the quality of the sensor orientation data. When the system is not fully calibrated, the orientation estimation can be bad. The calibration status is a 8-bit variable where every 2 bits represent a different piece of calibration info.

2.4.5 What's next?

You're at the end of the standard tutorials that introduce you to all of Pozyx's basic functionality: ranging, positioning, and its IMU. If you haven't already done so and have the number of devices necessary, you can read the multitag tutorial. When you start work on your own prototype, don't be afraid to delve into our documentation which should suffice for all your needs.

There's also the chat room tutorial, for which you will need two Arduinos and two Pozyx tags, in which you will build a small chat room where you use Pozyx to communicate between two computers. This opens the door to adding real communication and control to two Arduino-enabled Pozyx tags, as you can send and interpret Arduino commands this way as well.

2.5 Tutorial 4: Multitag positioning (Arduino)

2.5.1 Multitag positioning

The multitag positioning is an extension of tutorial 2, where we went over the positioning of a single device, and this being an extension is also visible in the code. Therefore, this tutorial will seem rather short and will go over the additions to allow for multitag. It is thus essential to follow that tutorial first.

For this tutorial you'll need an Arduino, at least four anchors and at least three other powered devices. These can be either tags or anchors, but remember that anchors don't have an IMU.

Open up the multitag positioning example in the Arduino IDE under File > Examples > Pozyx > multitag_positioning.

2.5.2 Plug and play

Looking at the parameters of the multitag positioning, you will see one major addition over the regular positioning example, namely the tags parameter which replaced the remote_id and remote parameters. In this list, you'll have to write the IDs of the remote devices you'll be using. For your anchors, use the setup you measured out in the positioning tutorial. You can also again choose the dimension and algorithm used for the positioning.

There is also the num_tags parameter, which you'll need to change to the size of your tags array.

```
int num_tags = 3;
uint16_t tags[3] = {0x0001, 0x0002, 0x0003};
```

Once you put the IDs of your tags here, you can start with multitag positioning. You can use the positioning Processing sketch straight from the bat as well if you set your use_processing to true and set your Processing serial and port correctly. However, reading the terminal output offers a more deterministic way of knowing whether you've got it working.

2.5.3 Code additions and changes

As this is an extension of the ready to localize tutorial, the code has been modified to support multiple remote devices through the added tags parameter. Printing both error and position now takes an additional parameter: the ID of the device we're working with.

2.5.3.1 Anchor configuration

Each device needs to be configured with the anchors to successfully perform positioning. The anchor configuration is performed on every device using setAnchorsManual, which iterates over every remote device, and feedback on this configuration is printed.

```

1 void setAnchorsManual(){
2   for (int i = 0; i < num_tags; i++){
3     int status = Pozyx.clearDevices(tags[i]);
4     for(int i = 0; i < num_anchors; i++){
5       device_coordinates_t anchor;
6       anchor.network_id = anchors[i];
7       anchor.flag = 0x1;
8       anchor.pos.x = anchors_x[i];
9       anchor.pos.y = anchors_y[i];
10      anchor.pos.z = heights[i];
11      status &= Pozyx.addDevice(anchor, tags[i]);
12    }
13    if (status == POZYX_SUCCESS){
14      Serial.print("Configuring ID 0x");
15      Serial.print(tags[i], HEX);
16      Serial.println(" success!");
17    }else{
18      printErrorCode("configuration", tags[i]);
19    }
20  }
21 }

```

2.5.3.2 Positioning loop

Positioning is done in the same way as the anchor configuration: every tag gets its turn. If successful, the position is printed and otherwise an error message gets printed. This is effectively a TDMA approach, where the remote devices won't interfere with each other, unlike if you perform positioning on different devices at the same time, which will not work.

```

1 void loop(){
2   for (int i = 0; i < num_tags; i++){
3     coordinates_t position;
4     int status = Pozyx.doRemotePositioning(tags[i], &position, dimension, height, algorithm);
5     if (status == POZYX_SUCCESS){
6       // prints out the result
7       printCoordinates(position, tags[i]);
8     }else{
9       // prints out the error code
10      printErrorCode("positioning", tags[i]);
11    }
12  }
13 }

```

2.5.4 Caveats and closing remarks

- Positioning multiple devices at once is performed in a TDMA manner so that they don't interfere with each other. If your devices crash, try adding a small delay between the devices positioning to add robustness.
- Since you're using TDMA, you're splitting the update rate between the different devices. If you'd start with 24 Hz for a single device, you'd get at most 12 Hz per device if you're using two devices on the same UWB settings. 8 Hz if you're using three, and so on. Depending on your usecase, this could be important.

- Having more devices in a line-of-sight of both your master device and your anchors might be harder depending on your use case. Account for this to smartly set up your anchors and master.
- If your area gets larger and you want to maintain a good update rate, adding more anchors which will loosen the constraints on your UWB settings is recommended.

Now that you finished all the tutorials, you should be all set for any usecase implementation that Pozyx is currently capable of. If you have any questions, remarks, or problems with the tutorials, please mail to support@pozyx.io¹¹

2.6 Chat room (Arduino)

Open up the chat room example in the Arduino IDE under File > Examples > Pozyx > chat_room. This example requires two or more pozyx shields and an equal number of Arduino's. In this example, it will be possible to chat wirelessly with all pozyx devices in range. Below is a figure of two devices talking. To run this demo, make sure that the serial monitor runs at 115200 baud with "both NL & CR" enabled.



Interpreting the LEDs: When this program is running, the Rx LED should be blinking, indicating that the Pozyx shield is listening for incoming data. Also LED1 should be blinking at a low frequency to indicate that the pozyx system is working properly. When this LED stops blinking, something went wrong. Finally, the TX led will blink shortly whenever you transmit a message.

¹¹ <mailto:support@pozyx.io>

2.6.1 The Arduino code explained

The Arduino sketch code begins as follows:

```

1  #include <Pozyx.h>
2  #include <Pozyx_definitions.h>
3  #include <Wire.h>
4  uint16_t source_id;           // the network id of this device
5  uint16_t destination_id = 0;  // the destination network id. 0 means the message is
    broadcasted to every device in range
6  String inputString = "";      // a string to hold incoming data
7  boolean stringComplete = false; // whether the string is complete

```

It can be seen that the sketch requires twopozyx files: "Pozyx.h" and "Pozyx_definitions.h" which contains the Pozyx Arduino library, and the "Wire.h" file for the I2C. After that, there are a number of global variables defined.

Next, we look at the setup() function. This function runs only once when the Arduino starts up.

```

1  void setup(){
2      Serial.begin(115200);
3      // initialize Pozyx
4      if(! Pozyx.begin(false, MODE_INTERRUPT, POZYX_INT_MASK_RX_DATA, 0)){
5          Serial.println("ERROR: Unable to connect to POZYX shield");
6          Serial.println("Reset required");
7          abort();
8      }
9      // read the network id of this device
10     Pozyx.regRead(POZYX_NETWORK_ID, (uint8_t*)&source_id, 2);
11     // reserve 100 bytes for the inputString:
12     inputString.reserve(100);
13     Serial.println("--- Pozyx Chat started ---");
14 }

```

From the code it can be seen that the Serial port is used at 115200 baud (~= bits per second). The other parameters for the serial port are default, i.e., 8bits, no parity, 1 stop bit. Next, the Pozyx device is initialized. The Pozyx.begin(boolean print_result, int mode, int interrupts, int interrupt_pin) function takes up to 4 parameters. This function checks that the Pozyx device is present and working correctly. The first parameter indicates if we want debug information printed out, the second describes the mode: MODE_POLLING or MODE_INTERRUPT. Using the polling mode, the Arduino will constantly poll (ask) the Pozyx shield if anything happened. Alternatively, the interrupt mode configures thepozyx device to give an interrupt signal every time an event has occurred (this is the preferred way). The events that should trigger in interrupt are configured by the interrupts parameter (more information about interrupts here). Lastly, with interrupt_pin it is possible to select between the two possible interrupt pins (digital pin 2 or 3 on the Arduino).

Here, thepozyx device is configured such that it doesn't output debug data, and that it uses interrupt using interrupt pin 0. With POZYX_INT_MASK_RX_DATA interrupts will be triggered each time wireless data is received.

Next, the setup function reads out the 16 bit network id of thepozyx shield (this is the hexadecimal number printed on the label), and stores it in the global variable source_id. Notice that the regRead function works with bytes, so we indicate that we want to read 2 bytes starting from the register address POZYX_NETWORK_ID and store those to bytes in the byte pointer. Finally, 100 bytes are reserved for the input string.

The code for the main loop is listed below:

```

1  void loop(){
2      // check if we received a newline character and if so, broadcast the inputString.
3      if(stringComplete){
4          Serial.print("0x");
5          Serial.print(source_id, HEX);
6          Serial.print(": ");
7          Serial.println(inputString);
8          int length = inputString.length();
9          uint8_t buffer[length];
10         inputString.getBytes(buffer, length);
11         // write the message to the transmit (TX) buffer
12         int status = Pozyx.writeTXBufferData(buffer, length);
13         // broadcast the contents of the TX buffer
14         status = Pozyx.sendTXBufferData(destination_id);
15         inputString = "";
16         stringComplete = false;
17     }
18     // we wait up to 50ms to see if we have received an incoming message (if so we receive an
19     RX_DATA interrupt)
20     if(Pozyx.waitForFlag(POZYX_INT_STATUS_RX_DATA, 50))
21     {
22         // we have received a message!
23         uint8_t length = 0;
24         uint16_t messenger = 0x00;
25         delay(1);
26         // Let's read out some information about the message (i.e., how many bytes did we receive and
27         who sent the message)
28         Pozyx.getLastDataLength(&length);
29         Pozyx.getLastNetworkId(&messenger);
30         char data[length];
31         // read the contents of the receive (RX) buffer, this is the message that was sent to this
32         device
33         Pozyx.readRXBufferData((uint8_t *) data, length);
34         Serial.print("0x");
35         Serial.print(messenger, HEX);
36         Serial.print(": ");
37         Serial.println(data);
38     }
39 }

```

In the main loop, the system waits for any of the following two events:

- **The user has written some text and pressed enter** which is checked by the if-statement `if(stringComplete)`. When this happened, the program will show the text in the user terminal and broadcast the text. The text is broadcasted using the following two statements: `Pozyx.writeTXBufferData(buffer, length);`, which puts the data in a transmit buffer, ready for transmission, and `Pozyx.sendTXBufferData(destination_id);` to actually transmit the data to the device with the given `destination_id`. In this example, `destination_id` is set to 0, meaning that the message is broadcasted (everyone will receive it).
- **The Pozyx device has received a wireless message.** This is checked by `Pozyx.waitForFlag(POZYX_INT_STATUS_RX_DATA, 50)`. The `waitForFlag` waits for the `RX_DATA` interrupt for a maximum of 50ms. If the interrupt happened, the function returns success. At this point, the length of the

message, and the network id of the device sending the message is obtained using `Pozyx.getLastDataLength(&length);` and `Pozyx.getLastNetworkId(&messenger);`. On the Pozyx device, the content of the received message is stored in the RX buffer. This content can be read using `Pozyx.readRXBufferData((uint8_t *) data, length);`.

This concludes the last example that is included in the Pozyx Arduino Library. Make sure to check out our Tutorials section where we will regularly post some new tutorials. If you have any requests for tutorials or if you have a tutorial of your own, please send us a message at info@pozyx.io¹².

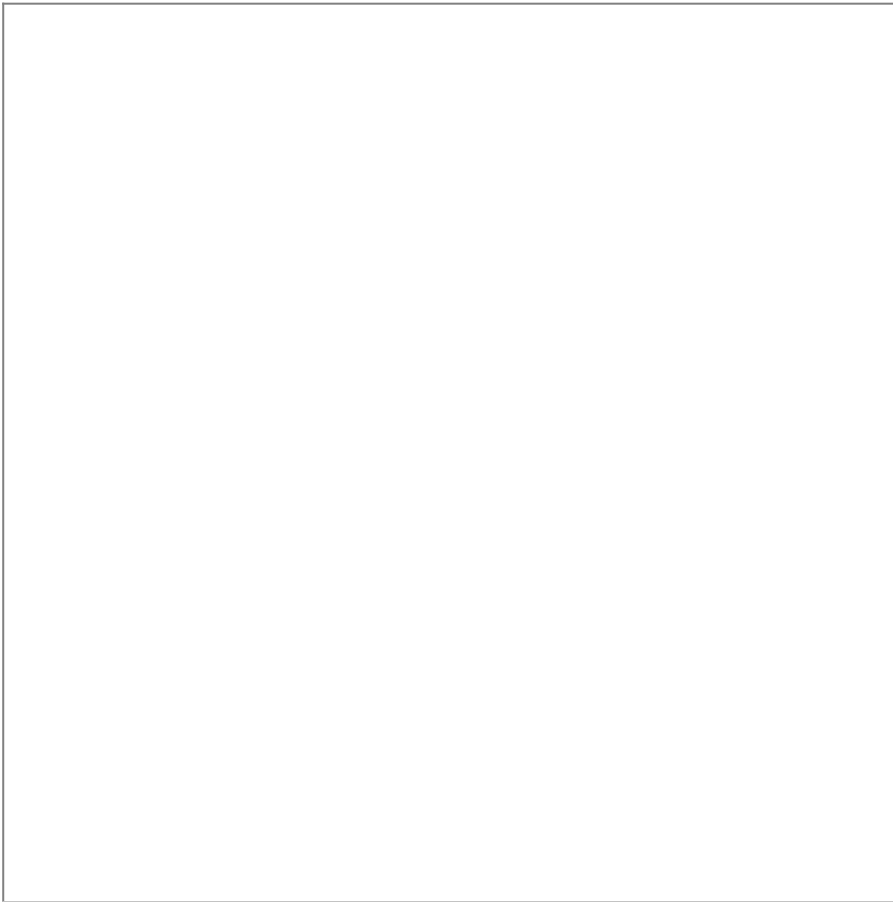
2.7 Configuration of the UWB parameters (Arduino)

2.7.1 The different UWB parameters

The selection of the ultra-wideband (UWB) parameters can be an important design choice. In this tutorial we explain how you can change the UWB settings and how this impacts the system's performance. Typically, the UWB parameters will impact the update rate, the range and the connectivity between devices. It is very important to realize that devices with different UWB settings may not be able to communicate with each other. Always make sure that all devices (tags and anchors) are using the same UWB settings in your application.

For this part, no coding is necessary as we will use one of the Arduino sketches included in the Pozyx library that will guide you through the configuration. Start by opening the sketch `pozyx_UWB_configurator` that can be found under `File > Examples > Pozyx > useful`. Upload the sketch to an Arduino equipped with a Pozyx shield and open the serial monitor. Once the sketch starts, it will search and show all Pozyx devices in range with their respective UWB configuration settings. This is an example output of the sketch as viewed in the serial monitor:

¹² <mailto:info@pozyx.io>




The UWB configurator has found 6 pozyx anchors, of which 5 are configured on channel 2 and, one is configured on channel 5.

After scanning, it is possible to select one of the devices and change its UWB settings. The new settings will be saved to the flash memory of the selected device and the device will keep using these settings even after restart. The following settings can be set:

- **channel** This sets the UWB channel. The pozyx device can use 6 independent UWB channels. Devices on different UWB channels cannot communicate and do not interfere with each other. In general, lower frequencies (i.e., lower channel numbers) also result in an increased communication range. More details can be found in the register description of POZYX_UWB_CHANNEL
- **bitrate** This sets the UWB bitrate. Three possible settings are possible: 110kbit/sec, 850kbit/sec and 6.81Mbit/sec. A higher bitrate will result in shorter messages and thus faster communication. However, this comes at the expense of a reduced operating range. The effect of the bitrate on the duration of ranging or positioning is shown in the figures below. More details can be found in the register description of POZYX_UWB_RATES
- **pulse repetition frequency (PRF)** This sets the UWB pulse repetition frequency. Two possible settings are possible: 16MHz or 64MHz. This setting has little effect on the communication rate. However, on the same UWB channel, these two settings can live next to each other without interfering. More details can be found in the register description of POZYX_UWB_RATES
- **preamble length** This sets the UWB preamble length. This setting has 8 different options: 4096, 2048, 1536, 1024, 512, 256, 128, or 64 symbols. A shorter preamble length results in shorter messages and thus faster communication. However, this again comes at the expense of a reduced operating range. The effect of the preamble length on the duration of ranging or positioning is shown in the figures below. More details can be found in the register description of POZYX_UWB_PLEN

From this, we can conclude that by changing the channel and PRF settings, up to 12 systems can run independently from each other. For the highest possible update rate (with the shortest range), the bitrate should be set to 6.81Mbit/sec and the preamble length to 64 symbols. Conversely, for the lowest possible update rate with the best range the bitrate should be set to 110kbit/sec with a preamble length of 4096 symbols, preferably on the first channel.

 Make sure to enter 'save' after reconfiguring each device with the UWB_configurator sketch to save the settings in flash.

2.7.2 Doing it in code

An overview of all the Arduino functions that can be used to manipulate the UWB settings in code can be found in the Arduino library documentation. Note that in order to save the settings for the next reset, it is required to use the function `saveConfiguration()`.

2.8 Changing the network ID (Arduino)

Ever since saving Pozyx registers to its flash memory was implemented, changing your device's ID has become possible. It's possible to do this both locally and remotely, and that's what will be discussed in this short article.

Open up the Arduino sketch in the Arduino IDE under File > Examples > Pozyx > useful > `pozyx_change_network_id`.

2.8.1 The code explained

2.8.1.1 Parameters

```
uint16_t new_id = 0x1000;    // the new network id of the pozyx device, change as desired
bool remote = true;         // whether to use the remote device
uint16_t remote_id = 0x6000; // the remote ID
```

The parameters are what you'd expect for changing the ID: the `new_id` the device will take, and whether we're working with a remote device with ID `remote_id`. Change these to suit your preferences. In the parameters shown above, running the code would mean that a remote device with ID 0x6000 would have its ID changed to 0x1000.

2.8.1.2 Changing the ID

Now, let's look at the code that effectively reconfigures a device's network ID.


```

void setup() {
  // ...
  // Pozyx and serial initialization
  Pozyx.setNetworkId(new_id, remote_id);
  uint8_t regs[1] = {POZYX_NETWORK_ID};
  status = Pozyx.saveConfiguration(POZYX_FLASH_REGS, regs, 1, remote_id);
  if(status == POZYX_SUCCESS){
    Serial.println("Saving to flash was successful! Resetting system...");
    Pozyx.resetSystem(remote_id);
  }else{
    Serial.println("Saving to flash was unsuccessful!");
  }
}

```

We can see that there are three steps involved in configuring a device with a new network ID. As of writing, `setNetworkId` doesn't change the network ID the Pozyx is using, and the Pozyx needs to be reset after its new ID has been saved to its flash memory. So these three steps are:

- Setting the Pozyx's new ID using `setNetworkId`
- Saving the Pozyx network ID register. Note that while the ID is a `uint16_t` value, we only use the first register address to save it. We don't need to save two registers.
- If saving was successful, we reset the device. After the reset, the device uses the new ID. All the while, textual feedback is given to the user.

2.8.2 Finishing up

Changing the ID does not seem to have an immediate purpose except for if you have devices with a double ID, but changing the ID can be convenient for other reasons as well to make your setup more maintainable. Identifying anchors with a prefix, for example. Naming your anchors 0xA001 up to 0xA004 or more. This is easier to read than the default device IDs. If you have anchors on different UWB channel, naming the set on channel 5 0xA501 to 0xA504 and the set on channel 2 0xA201-0xA204 will make your life easier, and so on.

2.9 Combination of cloud and Arduino

2.9.1 Introduction

The [companion software](https://www.pozyx.io/product-info/companion-software)¹³ is a great tool to manage your Pozyx setup of one or multiple tags. However, sometimes you still require to know the position on the tag itself. This is still perfectly possible. When a tag is remotely being positioned, it is still possible to read out the position and sensor data through the I2C or USB interface and integrate it on your Arduino.

In this tutorial, we show you how to read the position data using an Arduino connected to a slave tag that is **remotely positioned**. The master tag can be working with Python, Arduino or the Creator Controller, and this sketch is exclusive for slave tags.

2.9.2 Requirements

If you're working with Arduino, you should already have everything you need.

¹³ <https://www.pozyx.io/product-info/companion-software>

- One of our supported Arduinos: Uno, Mega or Nano with a slave tag
- Arduino Create: <https://create.arduino.cc>
- Pozyx Creator Controller/Arduino/Python with one master tag

2.9.3 Here we go!

- Make sure you have the latest version of the Pozyx Creator Controller running.
- Set up your anchors.
- Connect your tag, discover it and make sure you get good positioning coordinates.
- Install Arduino. We highly recommend the new Arduino create cloud app: <https://create.arduino.cc/editor>
- In Arduino create web editor, go to libraries and search for Pozyx. Clicking this will guide you to the Pozyx Github page. Navigate to the example sketch called "pozyx_check_new_position.ino" in the folder useful under examples.
- Plug your Arduino Uno or Mega into your computer, shield your Pozyx tag, and upload the Arduino sketch.
- Open the serial monitor at 115200 baud rate.
- Go back to the Arduino cloud app and discover the tag shielded to your Arduino. This will start the positioning of the remote tag.
- You now get a steady stream of positioning coordinates on your Arduino and on the serial monitor. You access these as "position.x", "position.y" and "position.z".

The following Arduino script reads out the position whenever it is available on the tag

```

1  #include <Pozyx.h>
2  #include <Pozyx_definitions.h>
3  #include <Wire.h>
4  void setup(){
5      Serial.begin(115200);
6      if(Pozyx.begin() == POZYX_FAILURE){
7          Serial.println(F("ERROR: Unable to connect to POZYX shield"));
8          Serial.println(F("Reset required"));
9          delay(100);
10         abort();
11     }
12 }
13 void loop(){
14     coordinates_t position;
15     int status = checkLocalNewPosition(&position);
16     if (status == POZYX_SUCCESS){
17         // prints out the result
18         printCoordinates(position);
19     }else{
20         // prints out the error code
21         printErrorCode("positioning");
22     }
23 }
24 int checkLocalNewPosition(coordinates_t *position)
25 {
26     assert(position != NULL);
27     int status;
28     uint8_t int_status = 0;
29     // now wait for the positioning to finish or generate an error
30     if (Pozyx.waitForFlag_safe(POZYX_INT_STATUS_POS | POZYX_INT_STATUS_ERR, 2*POZYX_DELAY_INTERRUPT,
&int_status)){
31         if((int_status & POZYX_INT_STATUS_ERR) == POZYX_INT_STATUS_ERR)
32         {
33             // An error occurred during positioning.
34             // Please read out the register POZYX_ERRORCODE to obtain more information about the error
35             return POZYX_FAILURE;
36         }else{
37             status = Pozyx.getCoordinates(position);
38             return status;
39         }
40     }else{
41         return POZYX_TIMEOUT;
42     }
43 }
44 // prints the coordinates for either humans or for processing
45 void printCoordinates(coordinates_t coor){
46     Serial.print("POS");
47     Serial.print(", x(mm): ");
48     Serial.print(coor.x);
49     Serial.print(", y(mm): ");
50     Serial.print(coor.y);
51     Serial.print(", z(mm): ");
52     Serial.println(coor.z);
53 }
54 // error printing function for debugging

```

```

55 void printErrorCode(String operation){
56     uint8_t error_code;
57     Pozyx.getErrorCode(&error_code);
58     Serial.print("ERROR ");
59     Serial.print(operation);
60     Serial.print(", local error code: 0x");
61     Serial.println(error_code, HEX);
62 }

```

As we already perform positioning via the master tag, the system checks whether new positions are available without performing the doPositioning command.

2.10 Troubleshoot basics (Arduino)

2.10.1 Overview

In this article, some basic code snippets are provided that can help you figure out any problems when things aren't going as they should. If you haven't read the getting started, please do so and install the necessary tools and libraries.

These code snippets can be found in a Arduino sketch as well, which can be used for troubleshooting a local and remote device straight from the bat. You can find this file in File > Examples > Pozyx > useful >pozyx_basic_troubleshooting.

We take the code snippet approach in this article because being able to re-use these in your own projects is the ultimate goal here. Here is an overview of the troubleshooting basics:

- Device check
- Network check
- Interpreting the LEDs

2.10.2 Local and remote device check

The first thing to do to get an insight in your Pozyx's operation status, is reading out its basic status registers.

Registers can be read using the function Pozyx.regRead() for your local device, or Pozyx.remoteRegRead() for a remote device. A full overview of all the registers with their description can be found here. Let's read out the first 5 bytes starting from the memory address POZYX_WHO_AM_I and display the result.

```
uint8_t data[5] = {0,0,0,0,0};
if (remote_id == NULL){
    Pozyx.regRead(POZYX_WHO_AM_I, data, 5);
    Serial.println("local device:");
}else{
    Pozyx.remoteRegRead(remote_id, POZYX_WHO_AM_I, data, 5);
    Serial.print("device 0x");
    Serial.println(remote_id, HEX);
}
Serial.print("who am i: 0x");
Serial.println(data[0], HEX);
Serial.print("firmware version: 0x");
Serial.println(data[1], HEX);
Serial.print("hardware version: 0x");
Serial.println(data[2], HEX);
Serial.print("self test result: 0b");
Serial.println(data[3], BIN);
Serial.print("error: 0x");
Serial.println(data[4], HEX);
```

For a tag, the output should show the following result:

```
who am i: 0x43
firmware version: 0x10
hardware version: 0x23
self test result: 0b111111
error: 0
```

From these results you can already learn a great deal:

- **who am i** is wrong: the Pozyx device is not running or it is badly connected with the Arduino. Make sure that the jumper is on the BOOT0 pins.
- **firmware version** is wrong: you must update the firmware as explained in updating the firmware. Make sure that all devices, anchors and tags are running on the same firmware version.
- **hardware version** is wrong: is your device on fire?
- **self test** is different: for the tag the result should be 0b111111, for the anchor it should be 0b110000. Check out POZYX_ST_RESULT for more information. Note: in firmware version 1.0, the self-test might currently display 0b110000 instead of 0b111111. This is an anomaly which occurs when the selftest is done before the sensors are initialized. Don't panic, this doesn't necessarily mean the sensors aren't initialized. You can try the third tutorial to quickly see whether the IMU is actually operational. Check if the problem remains after resetting the device.
- **the error code is not 0**: Something went wrong, this isn't necessarily dramatic. Check out POZYX_ERRORCODE to see which error was triggered. If you see error 0x09, you can safely ignore this.

Note that the function Pozyx.begin() will check most of these registers as well to see if everything is working properly.

2.10.2.1 Not finding a remote device

When the who am i is not correct when reading out the register remotely, there may be additional causes:

- The remote address is wrong. It is printed on the label on the device.
- The devices are not within range or their signal is blocked for some reason.

- The devices are configured with different UWB settings. The UWB settings must be the same to enable communication. See POZYX_UWB_CHANNEL and the subsequent registers.
- Multiple Pozyx device are transmitting at the same time and are interfering with one-another.
- If you're using doDiscovery to find devices, as in the UWB configurator (Arduino), the slot duration might be too short for the UWB settings of the devices. This happens at channel 1, max preamble length.

2.10.3 Network check

For positioning, a number of tags and anchors are required. Despite the hardware differences, each device can operate in both modes, for positioning. Anchors won't have IMU data. This operation mode is selected with jumper on the T/A pins.

- **Tag mode:** the jumper is present. This device can move around and perform positioning
- **Anchor mode:** the jumper absent. This device must remain stationary and will support other devices in positioning.

With the function Pozyx.doDiscovery() it is possible to obtain a list of all the devices within range. The function takes one parameter to discover either tags, anchors or both. The constants for this parameter is POZYX_DISCOVERY_ALL_DEVICES for all devices, POZYX_DISCOVERY_TAGS_ONLY for finding only tags and, as you've likely guessed, POZYX_DISCOVERY_ANCHORS_ONLY for anchors. The following code will find and display all devices within range:

```
Pozyx.clearDevices();
if( Pozyx.doDiscovery(POZYX_DISCOVERY_ALL_DEVICES) == POZYX_SUCCESS){
    uint8_t num_devices = 0;
    Pozyx.getDeviceListSize(&num_devices);
    Serial.print("Discovery found: ");
    Serial.print(num_devices);
    Serial.println(" device(s).");
    uint16_t tags[num_devices];
    Pozyx.getDeviceIds(tags, num_devices);
    for(int i = 0; i < num_devices; i++){
        Serial.print("0x");
        Serial.print(tags[i], HEX);
        if (i != num_devices - 1){
            Serial.print(", ");
        }
    }
    Serial.println();
}
```

2.10.4 Interpreting the LEDs

The Pozyx device has a number of status LEDs that can also be used to analyze the Pozyx behavior at a glance.

- LED 1: This LED should be blinking slowly to indicate that the system is responsive.
- LED 2: This LED indicates that the Pozyx device is performing a specific function (for example, calibration or positioning).
- LED 3: This LED has no function.
- LED 4: This LED will indicate that there was an error somewhere. The error can be found by reading the register POZYX_ERRORCODE.
- RX LED: This indicates that the UWB receiver is enabled and that the device can receive wireless signals.

- TX LED: This LED indicates that the device has transmitted a wireless signal.

Using the register `POZYX_CONFIG_LEDS` the status LEDs can be configured to be turned off.

3 Python

Connect to the Pozyx devices through USB with the Python Library available on GitHub and PyPi.

3.1 Downloads

- The **Pozyx Python Library**. The library can be obtained from [Github](#)¹⁴ or [downloaded from the zip file directly](#)¹⁵.
- The **Pozyx Python Library documentation** can be found here: <https://pypozyx.readthedocs.io/>
- [Tutorial 1: Ready to range \(Python\)](#)(see page 80)
- [Tutorial 2: Ready to localize \(Python\)](#)(see page 86)
- [Tutorial 3: Orientation 3D \(Python\)](#)(see page 95)
- [Tutorial 4: Multitag positioning \(Python\)](#)(see page 99)
- [Changing the network ID \(Python\)](#)(see page 100)
- [Troubleshoot basics \(Python\)](#)(see page 102)

3.2 Tutorial 1: Ready to range (Python)

3.2.1 Ready to range

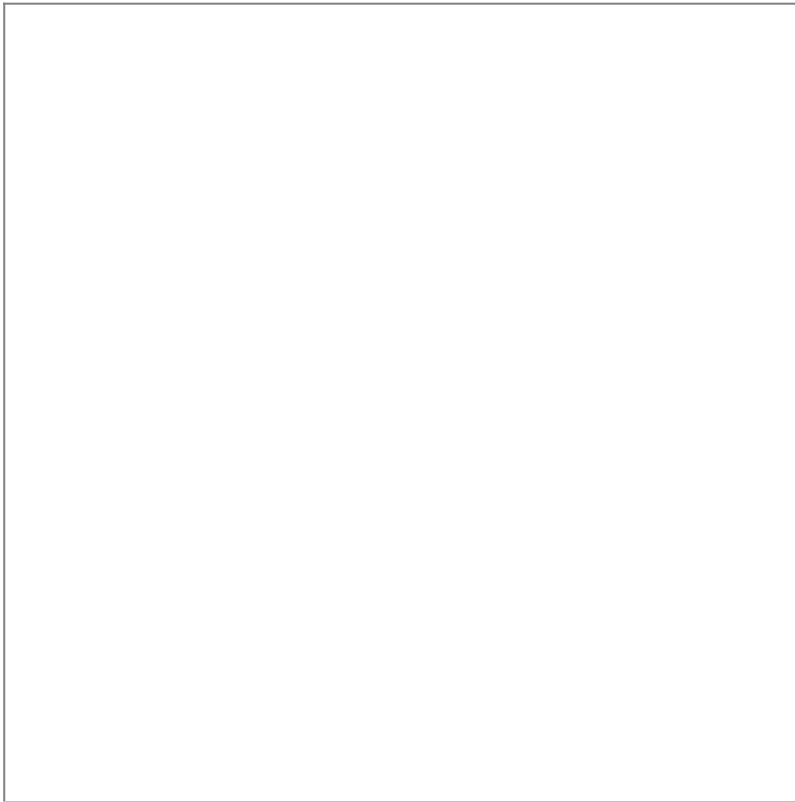
This is the first general Pozyx tutorial. If you haven't read the getting started, please do so and install the necessary tools and libraries.

This example requires two Pozyx devices. If all tools are installed, open up `ready_to_range.py` in the Pozyx library's tutorial folder. Probably, the path to this file will be "Downloads/Pozyx-Python-library/tutorials/ready_to_range.py". You can run the script from either command line or a text editor that allows running the scripts as well. If you're on Windows and have installed Python, you might be able to simply double-click it.

In this example, the distance between the two devices is measured and the onboard LEDs will show in which range the devices are to each other. The destination Pozyx's LEDs will be doing the same, commanded by our local Pozyx. The LEDs' changes with respect to the distance is sketched in the figure below:

¹⁴ <https://github.com/pozyxLabs/Pozyx-Python-library>

¹⁵ <https://github.com/pozyxLabs/Pozyx-Python-library/archive/master.zip>



3.2.2 Plug and play

To see the tutorial in action before we delve into the code, the parameters will need to be changed to match your destination device's ID.

At the bottom of the Python script, in the ifmain structure, we find the script's parameters. You can see that the first one is the serial port address of your device. Finding the serial port is described in getting started, but you can automate the port selection using `serial_port = get_serial_ports()[0].device`. This automatic serial port selection will be used in the next examples.

```
if __name__ == "__main__":
    port = 'COM1'          # COM port of the Pozyx device
    remote_id = 0x605D      # the network ID of the remote device
    remote = False         # whether to use the given remote device for ranging
    if not remote:
        remote_id = None
    destination_id = 0x1000 # network ID of the ranging destination
    range_step_mm = 1000   # distance between each LED lighting up.
    ranging_protocol = POZYX_RANGE_PROTOCOL_PRECISION #ranging protocol
```

You'll have to change the `destination_id` parameter to the ID of your destination Pozyx. Optionally, you can change the `range_step_mm` as well, which will make the distance range indicated by the LEDs either higher or lower depending on your change. `remote_id` and `remote` will allow measuring the distance between two remote Pozyx devices, and will be explained at the end of this tutorial, so don't change those for now.

The `ranging_protocol` is a new feature that allows changing between two ranging protocols: `POZYX_RANGE_PROTOCOL_FAST` and `POZYX_RANGE_PROTOCOL_PRECISION`. Precision is slower, but more precise and can be used on longer ranges. The fast protocol needs to warm up for about 100ms before. If this has piqued your interest, definitely check the system performance section in the datasheet!

If you've correctly changed your device ID, run the example! You should see that the LEDs of your two Pozyx devices now burn identically, and change when you move them closer or further apart. The output you're receiving will look like this:

```
16691ms, 7564mm, -93dBm
16753ms, 7718mm, -91dBm
16830ms, 7779mm, -92dBm
etc...
```

The first value is the device's timestamp of the range measurement. The second value then is the measured distance between both devices. The third value, expressed in dBm, signifies the signal strength, varying between -79 and -103 dB typically. You might be reading either:

```
0ms, 0mm, 0dB
or
ERROR: ranging
```

If this is the case, you probably miswrote the ID (did you write it with the 0x prefix?) or your device is not in range currently. As the LEDs will only indicate differences of a meter by default, keeping the distance between a couple of meters for this first try is recommended. If the error persists despite having put in everything correctly, check out the troubleshooting guide.

3.2.3 The code explained

Now that you've had a taste of what exactly this example provides, let's look at the code that provides this functionality. We'll begin with the imports and the setup functions, and then go into the continuous loop.

3.2.3.1 Imports

```
from pypozyx import *
```

We import everything from `pypozyx`. This import is made to provide you with the Pozyx constants, data containers, and the `PozyxSerial` class. This allows for general Pozyx use, as you have every library function available to you this way through `PozyxSerial`. If you need access to the register addresses, you can import `pypozyx.constants.registers`. For the bitmasks, import `pypozyx.constants.bitmasks`.

3.2.3.2 Emulating the Arduino flow

As a Python script isn't executed as an Arduino sketch, we emulate this with the following code. The `setup` function is run once, after which the `loop` function is executed continuously. We also pass the `PozyxSerial` object as a parameter, which makes the Pozyx not exclusive to this class if we want to add other functionality elsewhere. The other parameters defined above are passed, this has the purpose of not making this `ReadyToRange` class exclusive to this script, and stimulates reuse.

```
pozyx = PozyxSerial(port)
r = ReadyToRange(pozyx, destination_id, range_step_mm, remote_id)
r.setup()
while True:
    r.loop()
```

3.2.3.3 Setup

This function will be run only once at the script's execution, as you can see in the ifmain section, after this the loop will be continuously called. The PozyxSerial is initialized with the port we supplied in the parameters, so it's important to provide the correct port, as you'll notice when trying to run the code with an incorrect port. This initialization also checks for correct operation of the connected Pozyx.

```
def setup(self):
    """Sets up both the ranging and destination Pozyx's LED configuration"""
    print("-----POZYX RANGING V1.0 - -----")
    print("NOTES: ")
    print(" - Change the parameters: ")
    print("\tdestination_id(target device)")
    print("\trange_step(mm)")
    print()
    print("- Approach target device to see range and")
    print("led control")
    print("- -----POZYX RANGING V1.0 -----")
    print()
    print("START Ranging: ")
    # make sure the local/remote pozyx system has no control over the LEDs.
    led_config = 0x0
    self.pozyx.setLedConfig(led_config, self.remote_id)
    # do the same for the destination.
    self.pozyx.setLedConfig(led_config, self.destination_id)
```

Depending on the value of the remote parameters, we set the remote_id to use either the local or a remote Pozyx. After initializing the Pozyx, we configure both the local and destination Pozyx's LEDs to be in our control instead of displaying system status, which is their default behavior. This is done by setting the POZYX_CONFIG_LEDS register's value to 0b00000000 on both devices. While the library's functionality hides away a lot of low-level functionality, it's important to know how to access registers directly low-level and to find your way in the register overview for your own personal goals. These core low-level functions are:

```
getRead(address, data, remote_id=None)
setWrite(address, data, remote_id=None)
useFunction(function, params=None, data=None, remote_id=None)
```

3.2.3.4 Loop

Let's move on to the loop function, which will run continuously and executes the ranging functionality.

```
def loop(self):
    """Performs ranging and sets the LEDs accordingly"""
    device_range = DeviceRange()
    status = self.pozyx.doRanging(self.destination_id, device_range, self.remote_id)
    if status == POZYX_SUCCESS:
        print(device_range)
        if self.ledControl(device_range.distance) == POZYX_FAILURE:
            print("ERROR: setting (remote) leds")
    else:
        print("ERROR: ranging")
```

In the loop function, ranging is performed with the Pozyx library's doRanging function. The range information will be contained in device_range.

The device_range variable is an instance of the DeviceRange class. Looking at its definition (in device.py in the pypozyx/structures folder, we can see that the device range consists of three variables providing information about the measurement. The library also provides human-readable string conversion of its important data structures, and print(device_range) prints the range data in our desired format. The ByteStructure parent class provides functionality that makes sure the structure's bytes are packed together.

```
class DeviceRange(ByteStructure):
    def __init__(self, timestamp=0, distance=0, RSS=0):
        """Initializes the DeviceRange object."""
        self.timestamp = timestamp
        self.distance = distance
        self.RSS = RSS
```

In a wireless system, we don't want to waste transmission time on unused padding bytes, after all. The library provides similar structures for all relevant data structures related to Pozyx, which you'll encounter if you follow the other tutorials and get into further reading. The measurement consists of the timestamp in milliseconds, the measured distance in mm and the RSS-value (the received signal strength) expressed in dBm. Depending on the UWB settings, -103 dBm is more or less the lowest RSS at which reception is possible, and -80 is a normal value for devices close to each other. Upon a successful measurement, these three variables are printed and the LEDs are updated with the ledControl function.

3.2.3.5 ledControl

```
int ledControl(uint32_t range){
    int status = POZYX_SUCCESS;
    // set the LEDs of the pozyx device
    status &= Pozyx.setLed(4, (range < range_step_mm), remote_id);
    status &= Pozyx.setLed(3, (range < 2*range_step_mm), remote_id);
    status &= Pozyx.setLed(2, (range < 3*range_step_mm), remote_id);
    status &= Pozyx.setLed(1, (range < 4*range_step_mm), remote_id);
    // set the LEDs of the destination pozyx device
    status &= Pozyx.setLed(4, (range < range_step_mm), destination_id);
    status &= Pozyx.setLed(3, (range < 2*range_step_mm), destination_id);
    status &= Pozyx.setLed(2, (range < 3*range_step_mm), destination_id);
    status &= Pozyx.setLed(1, (range < 4*range_step_mm), destination_id);
    // status will be zero if setting the LEDs failed somewhere along the way
    return status;
}
```

```
def ledControl(self, distance):
    """Sets LEDs according to the distance between two devices"""
    status = POZYX_SUCCESS
    ids = [self.remote_id, self.destination_id]
    # set the leds of both local/remote and destination pozyx device
    for id in ids:
        status &= self.pozyx.setLed(4, (distance < range_step_mm), id)
        status &= self.pozyx.setLed(3, (distance < 2 * range_step_mm), id)
        status &= self.pozyx.setLed(2, (distance < 3 * range_step_mm), id)
        status &= self.pozyx.setLed(1, (distance < 4 * range_step_mm), id)
    return status
```

The `ledControl` function turns the LEDs on or off if they fall within a certain range, using a simple boolean expression, for example `range < 3*range_step_mm`, which returns true if the distance is within three times the `range_step_mm` parameter.

Nothing is stopping you from designing your own `ledControl` function, where you can use other indexes or, as a challenge, even use the four LEDs as 4 bits indicating up to sixteen `range_step_mm` differences instead of four.

3.2.3.6 Remote operation

As you have probably seen when looking at the code and the register functions, Pozyx is capable of communicating with remote Pozyx devices: thanks to UWB communication, we can perform every local functionality on a remote device as well. In this tutorial, this can be done by changing the `remote_id` and `remote` parameters. Setting `remote = true`; will perform all the functionality on a device with ID `remote_id`, so you will need to set that ID correctly as well and own three devices to perform remote ranging.

A general rule with the Pozyx API is that almost every function has `remote_id` as a keyword argument, which allows the writing of code that can easily switch between being executed locally or remotely, as seen in this example. When `remote_id` is uninitialized, the function will be executed locally.

The disadvantages of working remotely need to be kept in mind, however. The possibility of losing connection, something that is much harder with a cable. A delay, as the UWB communication between both devices is not

instantaneous. And you need a local controller, which is an extra device in your setup. In turn, you gain a lot of mobility, as the remote device only needs to be powered, not needing a controller.

This concludes the first Pozyx example for Python. In the next example, we'll cover positioning and the necessary setup to perform positioning successfully.

3.3 Tutorial 2: Ready to localize (Python)

3.3.1 Ready to localize

This is the second Pozyx tutorial, in which we'll go through the process of performing (remote) positioning with the Pozyx system. If you missed the first one, check that one out first, as each tutorial assumes knowledge of the concepts explained in the ones before.

For this example, you need to own at least the contents of the Ready to Localize kit. If all tools are installed, open up `ready_to_localize.py` in the Pozyx library's tutorial folder. Probably, the path to this file will be "Downloads/Pozyx-Python-library/tutorials/ready_to_localize.py". You can run the script from either command line or a text editor that allows running the scripts as well. If you're on Windows and have installed Python, you might be able to simply double-click it. If you get an error, you likely forgot to install `pythonosc`, you can do this easily using

```
pip install python-osc
```

In this example, we will first set up and measure the anchor locations to perform positioning, after which we'll be able to get accurate position data of the Pozyx, relative to the anchor setup. We will go in detail through other Pozyx core concepts as well, such as how to check the Pozyx's device list, and how to read its error status. Mastering these debugging tools now will make using Pozyx in your own projects easier.

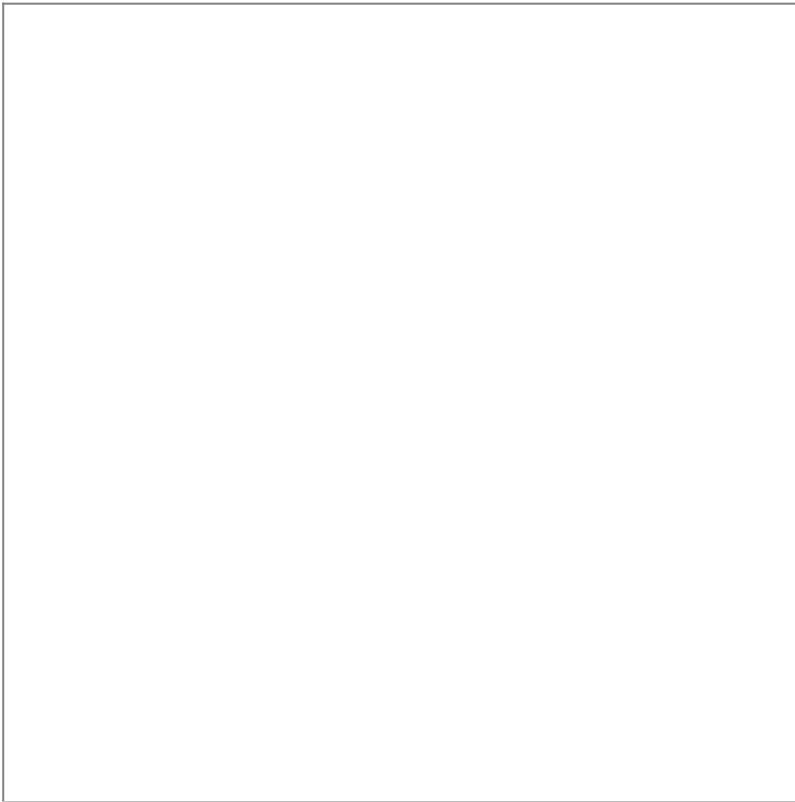
3.3.2 Anchor setup and measurement

3.3.2.1 Anchor setup

The Pozyx positioning system requires that the four anchors are placed inside the area where you wish to do positioning. In the guide 'Where to place the anchors?', it is explained how to place the anchors for the best possible positioning accuracy. The rules of thumb for the anchor placement were:

- Place the anchors high and in line-of-sight of the user.
- Spread the anchors around the user, never all on a straight line!
- Place the anchors vertically with the antenna at the top, power plug at the bottom.
- For 3D positioning: place the anchors at different heights.

It's also important to keep metallic objects out of immediate range of the antennas.



Before you install the anchors (with the provided Velcros or screws) on the walls or ceiling, it is usually a good idea to make a small sketch of the room, writing down the anchors' IDs and sketching where they will be placed. You can find the network ID as the hexadecimal number on the label adorning your anchor.

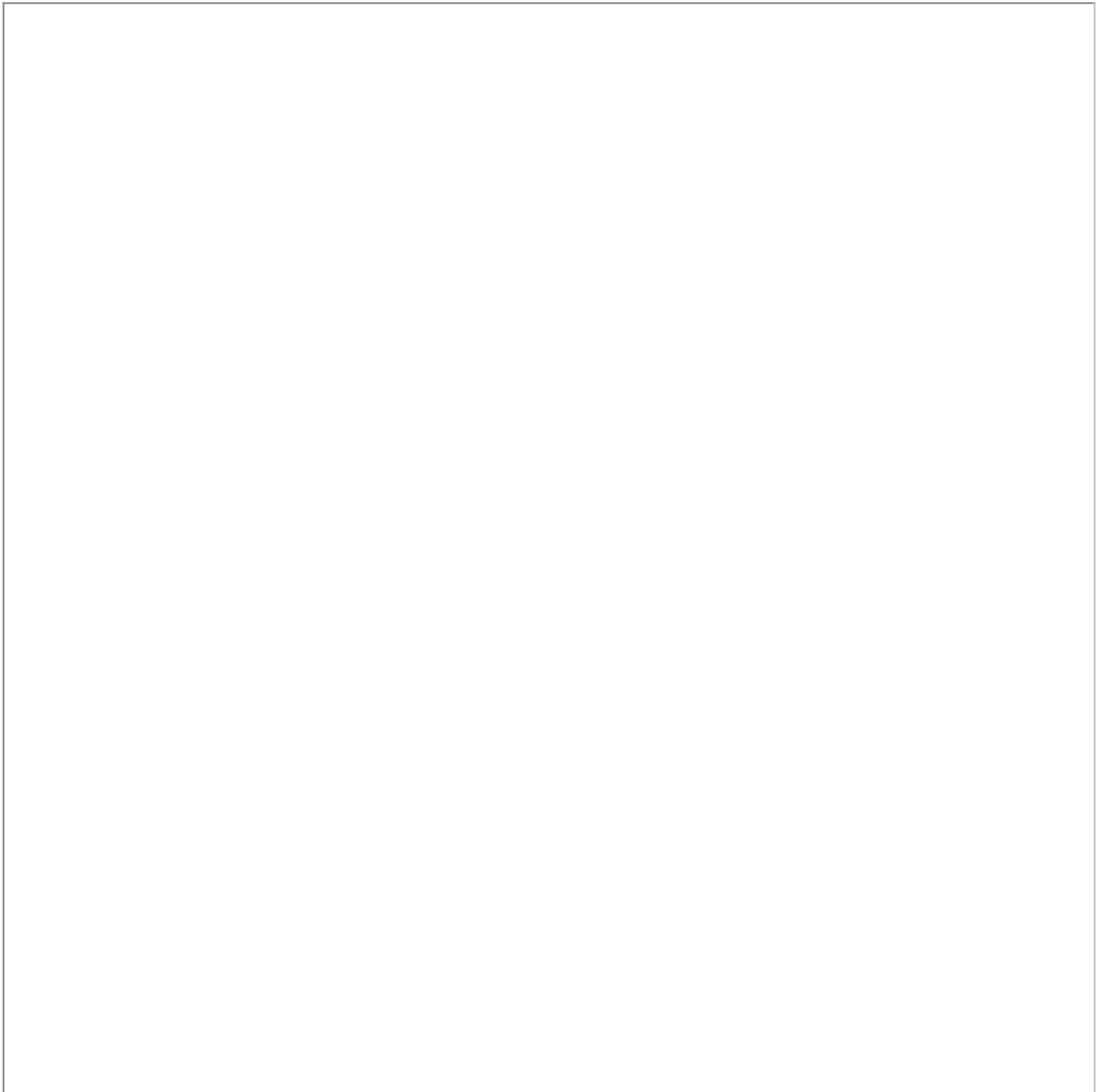
Remember that, for optimal antenna performance, it is recommended to place the anchors vertically with their antenna at the top, and to orient your tags that will be positioning vertically as well. Also make sure that no heavy metal objects are placed near the antenna, as this might degrade performance. We stress this again because it's just that important.

3.3.2.2 Measurement

While there are different ways to measure your distance, if you are serious about using Pozyx we recommend to have a laser measurer. This will be much more convenient, and much more accurate, than trying to measure out several meters between your anchors with a conventional foldable or extendable We recommend placing them in an approximate rectangle as in the image above. This allows you to simplify both measurement and setup, and to follow along better with the example.

3.3.2.3 The Pozyx coordinate system

Pozyx determines its position relative to the anchor coordinates you supplied it with. This gives you the freedom to position and rotate your axis in any way you want. This is shown in the images below:



You can see that the coordinate system on the left, which is conventional to the room's shape and orientation, is made by using anchor 0x256B as an element of the x-axis, giving it a zero y-coordinate. Then the y-axis is perpendicular to this, and anchor 0x3325 therefore has a non-zero x-component to fit into this orthogonal system. Anchor 0x1156 is selected as the origin, again for convenience. This origin doesn't need to be one of the anchors, however, and you could use the center of the room as the origin point just as well, modifying the anchor's coordinates accordingly: anchors 0x1156 and 0x3325 would have a negative x-component, while 0x2568 and 0x4244 have a positive coordinate. In turn, 0x1156 and 0x256B have a negative y-component while 0x3325 and 0x4244 will have a positive one.

In the right example, the origin is chosen to be anchor 0x3325's location, and the x-axis is matched with the path between anchors 0x3325 and 0x1156.

Let's go over the plan of attack in measuring out your setup, following the approach shown on the left:

- Use the antenna's center as Pozyx's position in space as best as you can.
- Pick one of your anchors as the origin, like we did 0x1156.
- Pick the anchor on the same wall to be the one defining your x-axis. We picked 0x2568.
- Measure out the horizontal distance between both anchors. Not a direct distance as this will include the vertical distance as well (Pythagoras). This will be that anchor's x-coordinate. In our case, this was 4.5 m.
- Now measure the distance to the opposite wall. Mark this point on the wall, as this is the x-axis's zero value on that wall. If both walls are parallel, and there are anchors attached directly to this wall, you can set their y-coordinate to this distance. In our case, we could set 0x4244's y-coordinate directly to 3.5 m.
- Now measure the x-coordinates of every anchor on that wall directly using the point you marked earlier, again assuming both walls are parallel. Measurements can be complicated if they are not, so use more reference points then.
- If there are anchors a bit apart from the wall, like 0x3325, be sure to account for this in its y-coordinate.

In this example, we've used the approach described above. We'll also assume the anchors are on heights between 1.1 and 2 meter, leading to these coordinates for each anchor:

0x1156: (0, 0, 1500)

0x256B: (4500, 0, 1800)

0x3325: (500, 3300, 1100)

0x4244: (4450, 3500, 2000)

These are the coordinates used in the example, but you can't copy these! You will have to change both anchor IDs and coordinates to match your own setup. All these coordinates are expressed in millimeters.

3.3.3 Plug and play

To get the example working and see what exactly is happening before we delve into the code, we'll need to change the parameters to match your device IDs and the coordinates you measured.

At the bottom of the Python script, in the ifmain structure, we find the script's parameters. You can see that the first one is the serial port address of your device. We use `serial_port = get_serial_ports()[0].device` to automate the port selection, and will do so from now on.

The anchors' data is created as a list of DeviceCoordinates objects. DeviceCoordinates' second parameter, POZYX_ANCHOR, is a flag indicating you're adding an anchor device. You'll have to match the IDs and coordinates to the ones of your setup. We've done that for the example setup from above, as a reference.

```

1  if __name__ == "__main__":
2      # shortcut to not have to find out the port yourself
3      serial_port = get_serial_ports()[0].device
4      remote_id = 0x1000          # remote device network ID
5      remote = False             # whether to use a remote device
6      if not remote:
7          remote_id = None
8      use_processing = False      # enable to send position data through OSC
9      ip = "127.0.0.1"           # IP for the OSC UDP
10     network_port = 8888         # network port for the OSC UDP
11     osc_udp_client = None
12     if use_processing:
13         osc_udp_client = SimpleUDPClient(ip, network_port)
14     # necessary data for calibration, change the IDs and coordinates yourself
15     anchors = [DeviceCoordinates(0x0001, 1, Coordinates(0, 0, 2000)),
16               DeviceCoordinates(0x0002, 1, Coordinates(3000, 0, 2000)),
17               DeviceCoordinates(0x0003, 1, Coordinates(0, 3000, 2000)),
18               DeviceCoordinates(0x0004, 1, Coordinates(3000, 3000, 2000))]
19     algorithm = POZYX_POS_ALG_UWB_ONLY # positioning algorithm to use
20     dimension = POZYX_3D              # positioning dimension
21     height = 1000                    # height of device, required in 2.5D positioning

```

You will also see the `remote_id`, `remote`, `algorithm`, `dimension`, and `use_processing` parameters. `remote_id` and `remote` should be familiar from the first tutorial. `use_processing` will be used for the visualization. `algorithm`, `dimension`, and `height` will allow for customization regarding the positioning, and we'll get back to these when we're looking at the code in detail. Leave these parameters unchanged for now, which will result in 3D positioning being done with the UWB-only algorithm.

Now that you've done all the plugging, it's time for play. Run the example, and if all goes well, you're now looking at coordinates filling up the window, after the manual anchor configuration is checked and printed. As you are using your local device, the ID will be set to 0.

```

POS ID 0x0000, x(mm): 1231 y(mm): 2354 z(mm): 1167
POS ID 0x0000, x(mm): 1236 y(mm): 2241 z(mm): 1150
etc...

```

That's that! You're now getting accurate position coordinates from the Pozyx! You might also be seeing one of the following:

```

POS ID 0x0000, x(mm): 0 y(mm): 0 z(mm): 0
or
ERROR configuration on ID 0x0000, error code 0xXX
or
ERROR positioning on ID 0x0000, error code 0xXX
or
Coordinates that are nothing even remotely close to what they should be

```

If so, you probably miswrote one of the anchor IDs or entered either wrong or mixed up coordinates, causing the positioning algorithm to fail. If the error persists despite having put in everything correctly, check out the troubleshooting guide. Now that you've seen the positioning in action, let's look at the code that made this possible.

3.3.4 The code explained

We will now cover the essential code to get positioning working, but there's a lot more code in the file. The extras segment will cover how to access the Pozyx's device list, how to retrieve error codes returned by the Pozyx, and how to retrieve additional sensor data each time you position.

3.3.4.1 Setup and manual calibration

The Pozyx serial connection is initialized, together with the ReadyToLocalize object, and its setup is called.

```

1  def setup(self):
2      """Sets up the Pozyx for positioning by calibrating its anchor list."""
3      print("-----POZYX POSITIONING V1.0 -----")
4      print("NOTES: ")
5      print("- No parameters required.")
6      print()
7      print("- System will auto start calibration")
8      print()
9      print("- System will auto start positioning")
10     print("-----POZYX POSITIONING V1.0 -----")
11     print()
12     print("START Ranging: ")
13     self.pozyx.clearDevices(self.remote_id)
14     self.setAnchorsManual()
15     self.printPublishConfigurationResult()

```

The setup function is straightforward. After its initialization of the Pozyx, it performs the anchor configuration. It first clears the Pozyx's device list using `clearDevices()`, and then manually adds the anchors in `setAnchorsManual()` we set up and measured out. Let's look at how this is done:

```

1  def setAnchorsManual(self):
2      """Adds the manually measured anchors to the Pozyx's device list one for one."""
3      status = self.pozyx.clearDevices(self.remote_id)
4      for anchor in self.anchors:
5          status &= self.pozyx.addDevice(anchor, self.remote_id)
6      if len(anchors) > 4:
7          status &= self.pozyx.setSelectionOfAnchors(POZYX_ANCHOR_SEL_AUTO, len(anchors))
8      return status

```

As each anchor is already defined as a `DeviceCoordinates` object, we can just iterate over the list of anchors and add each anchor to the device's stored device list, which it will use when positioning. If you'd use more than four anchors, the device's anchor selection is set to automatically use all available anchors from this set.

3.3.4.2 Loop

Now that we've properly configured the device's device list with our set of anchors, let's look at just how easy the actual positioning is in the short `loop()` function:

```

1  def loop(self):
2      """Performs positioning and displays/exports the results."""
3      position = Coordinates()
4      status = self.pozyx.doPositioning(
5          position, self.dimension, self.height, self.algorithm, remote_id=self.remote_id)
6      if status == POZYX_SUCCESS:
7          self.printPublishPosition(position)
8      else:
9          self.printPublishErrorCode("positioning")

```

We first create a new object that will contain the Pozyx's measured coordinates, after which we call the Pozyx's `doPositioning` function, which will perform the positioning algorithm and store the coordinates, measured respectively to the anchors, in the position object. That's essentially the entire positioning loop! If `doPositioning` returns `POZYX_SUCCESS`, the position will be printed in a human readable way. We see that we can pass the positioning's algorithm, dimension, and Pozyx height (used in 2.5D) as parameters in the positioning. There are three dimensions supported by Pozyx: `POZYX_2D`, `POZYX_2_5D`, and `POZYX_3D`.

In 2D the anchors and tags must all be located in the same horizontal plane. This is not the case for semi-3D or 3D positioning. In semi-3D positioning the height of the tag must be supplied in the height parameter (for example, when the tag is mounted on a driving robot with fixed height). The reason why semi-3D exists is because in many cases it is not possible to obtain an accurate estimate for the z-coordinate in 3D-positioning. This is a result of how the anchors are placed and is explained in the guide 'Where to place the anchors?'. As a final parameter you can supply which algorithm to use. Possible values are `POZYX_POS_ALG_UWB_ONLY` and `POZYX_POS_ALG_TRACKING`. By default `POZYX_POS_ALG_UWB_ONLY` is used. For more information about the algorithms we refer to `POZYX_POS_ALG`. We've defined the algorithm as a parameter, so you can change the algorithm in the parameters section instead of directly doing so in the function.

Some example usages:

- `status = Pozyx.doPositioning(&position, POZYX_2_5D, 1000)` semi-3D positioning with the height of the tag fixed at 1000mm (about 3.3feet).
- `status = Pozyx.doPositioning(&position, POZYX_3D)` 3D positioning, this requires at least 4 anchors. Note that the anchors must be placed at different heights to obtain a good accuracy of the z-coordinate.

3.3.5 Remote positioning

Positioning the Pozyx attached to your PC means that, except for when you have a long cable, you'll also need to be able to move your PC around if you want to position over a larger area than your desk. This would also mean that if you'd want to track objects, you'd need to attach a processing unit to these objects as well, instead of only the Pozyx. Luckily, the Pozyx attached to your computer can act as a master device, commanding one or more remote 'slaves'.

To position a remote device, you need to do the same as on a local device: put anchors in its device list that it will use for the positioning. A common misconception about the use of Pozyx is that configuring the anchors on your local device will make remote devices capable of positioning straight off the bat, but this isn't the case. You will notice that the `addDevice` function in `setAnchorsManual` adds the device to a remote device if remote positioning is enabled, thusly configuring the anchors on the remote device and not on the local one.

3.3.6 Extras

While not necessary for positioning, the added functionality in the code can go a long way for extending the use of Pozyx or when things go wrong, so going over these extras is recommended if you want to take things further without needing to figure things out yourself.

3.3.6.1 Printing the configuration result

To find out whether the calibration was in fact successful, we will retrieve the Pozyx's device list. This is the reverse of the configuration step, as we now retrieve the IDs and coordinates in turn from the Pozyx. Pozyx requires this to be done in several steps:

- Firstly we retrieve the size of the device list through `getDeviceListSize`.
- We use this size to create an appropriately sized device list container
- We retrieve the IDs in the device's device list using this container with `getDeviceIds`. You can also use `getTagIds` and `getAnchorIds` to have more control over which device IDs you're getting.
- Now we can get the device coordinates belonging to each of these IDs using `getDeviceCoordinates`

This is what is done in the code below, and we print the anchor's ID with its retrieved coordinates. If these don't match the anchors you passed to the device, something went wrong and it is recommended to try again. If this keeps failing, try going through the troubleshooting.

```

1  def printPublishConfigurationResult(self):
2      """Prints and potentially publishes the anchor configuration result in a human-readable
   way."""
3      list_size = SingleRegister()
4      status = self.pozyx.getDeviceListSize(list_size, self.remote_id)
5      print("List size: {0}".format(list_size[0]))
6      if list_size[0] != len(self.anchors):
7          self.printPublishErrorCode("configuration")
8          return
9      device_list = DeviceList(list_size=list_size[0])
10     status = self.pozyx.getDeviceIds(device_list, self.remote_id)
11     print("Calibration result:")
12     print("Anchors found: {0}".format(list_size[0]))
13     print("Anchor IDs: ", device_list)
14     for i in range(list_size[0]):
15         anchor_coordinates = Coordinates()
16         status = self.pozyx.getDeviceCoordinates(
17             device_list[i], anchor_coordinates, self.remote_id)
18         print("ANCHOR,0x%0.4x, %s" % (device_list[i], str(anchor_coordinates)))
19         if self.osc_udp_client is not None:
20             self.osc_udp_client.send_message(
21                 "/anchor", [device_list[i], int(anchor_coordinates.x), int(anchor_coordinates.y),
22                 int(anchor_coordinates.z)])
23         sleep(0.025)

```

3.3.6.2 Printing the error code

Pozyx's operation can misbehave due to various reasons. Other devices not being in range, being on different settings, or another firmware version... As you're getting started with Pozyx, it's hard to keep track of where exactly

things go wrong, and it's for this reason that Pozyx keeps track of what went wrong in the error status register, POZYX_ERRORCODE. The error code can be read in two ways, one uses the `getErrorCode` function to directly read out the value from the error register, while the other, through `getSystemError`, returns a more verbose error message, representing the relative error textually. For example, as the error code would be 0x05, `getSystemError` would return "Error 0x05: Error reading from a register from the I2C bus". While the latter is ideal when working in a printed environment, if you work with visualizations it's less than ideal to handle this entire string and you can perform custom functionality with the error code.

```

1  def printPublishErrorCode(self, operation):
2      """Prints the Pozyx's error and possibly sends it as a OSC packet"""
3      error_code = SingleRegister()
4      network_id = self.remote_id
5      if network_id is None:
6          self.pozyx.getErrorCode(error_code)
7          print("ERROR %s, local error code %s" % (operation, str(error_code)))
8          if self.osc_udp_client is not None:
9              self.osc_udp_client.send_message("/error", [operation, 0, error_code[0]])
10         return
11     status = self.pozyx.getErrorCode(error_code, self.remote_id)
12     if status == POZYX_SUCCESS:
13         print("ERROR %s on ID %s, error code %s" %
14               (operation, "0x%0.4x" % network_id, str(error_code)))
15         if self.osc_udp_client is not None:
16             self.osc_udp_client.send_message(
17                 "/error", [operation, network_id, error_code[0]])
18     else:
19         self.pozyx.getErrorCode(error_code)
20         print("ERROR %s, couldn't retrieve remote error code, local error code %s" %
21               (operation, str(error_code)))
22         if self.osc_udp_client is not None:
23             self.osc_udp_client.send_message("/error", [operation, 0, -1])

```

In this example, simple but comprehensive error checking is executed. If the positioning is purely local, the local error is read. When remote positioning fails, indicated by the positioning function returning `POZYX_FAILURE`, the error register is read. If the error couldn't be read remotely, the error is read locally. We are using `getErrorCode` instead of `getSystemError` because this allows us to efficiently send the error data to the visualization, and customize our error output. You can find the resulting error codes' meaning at the register documentation of `POZYX_ERRORCODE`.

3.3.6.3 Adding sensor information

Sensor information, such as orientation or acceleration, can easily be added to the code, as to return this sensor data every positioning loop. Adding orientation and/or acceleration allows you to get a better insight in the object you're tracking, but you'll have to account for the reduced positioning update rate caused by this additional operation. Especially remotely, this will delay your update rate. In this example code, not present in the actual script, we'll retrieve both orientation and acceleration

```
1 def printOrientationAcceleration(self):
2     orientation = EulerAngles()
3     acceleration = Acceleration()
4     self.pozyx.getEulerAngles_deg(orientation, self.remote_id)
5     self.pozyx.getAcceleration_mg(acceleration, self.remote_id)
6     print("Orientation: %s, acceleration: %s" % (str(orientation), str(acceleration)))
```

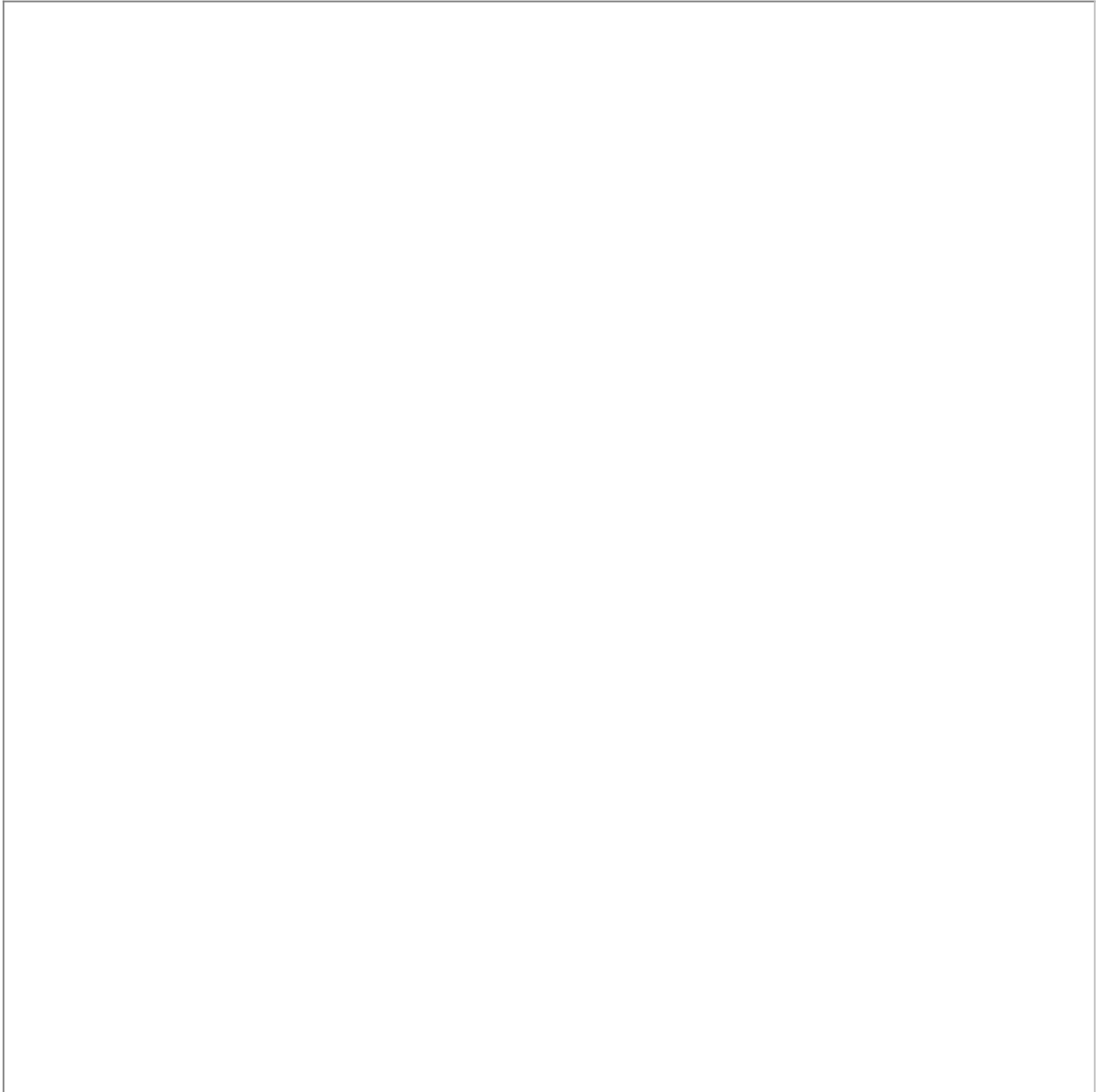
3.4 Tutorial 3: Orientation 3D (Python)

3.4.1 Downloads

In this example, the sensor data is read from the pozyx shield (either the shield mounted on the Arduino, or a remote shield). To visualize all the data, we use Processing. Below a screenshot of the Processing program showing all the sensor data graphically.

This example requires one or two pozyx shields. If all tools are installed, open up orientation_3D.py in the Pozyx library's tutorial folder. Probably, the path to this file will be "Downloads/Pozyx-Python-library/tutorials/orientation_3D.py". You can run the script from either command line or a text editor that allows running the scripts as well. If you're on Windows and have installed Python, you might be able to simply double-click it. If you get an error, you likely forgot to install pythonosc, you can do this easily using `pip install python-osc`.

Open up the pozyx_orientation3D.pde sketch in Processing, as this example will be directly visualized.



3.4.2 Plug and play

In Processing, make sure serial is set to false. Unless you need port 8888 for something else, you can leave both the Python script and Processing sketch intact. The data will automatically be sent using OSC on port 8888.

You should be able to physically rotate your Pozyx now and directly see it rotate in Processing as well. There are plots on the side and at the top, but what do they all mean?

3.4.3 Understanding the sensor data

- **Acceleration (g):** the acceleration is measured along 3 axes (i.e., in 3 directions) which is shown by three different colors on the plot. The acceleration is expressed in g's (from gravity). 1g is exactly the acceleration from earth's gravitation pull ($1g = 9.81m/s^2$). Because the acceleration from gravity is always present and equal to 1g it can be used to determine how the device is tilted or pitched. Try rotating the Pozyx device by 90 degrees (slowly) and you will see the the acceleration change from one axis to the other.
- **Magnetic field strength (μT):** the magnetic field strength is also measured along 3 axes and is expressed in μT . It can be used to measure the earth's magnetic field which varies between 30 and 60 μT over the earth's surface. The measured magnetic field strength can be used to estimate the magnetic north (similar to a compass). However, magnetic fields exist wherever electric current flows or magnetic materials are present. Because of this, they will influence the sensor data and it will become impossible to determine the magnetic north exactly. Try holding a magnet or something iron close to the pozyx device and you will see the magnetic field strength fluctuate.
- **Angular velocity (deg/s):** The angular velocity is measured by the gyroscope and measures how fast the device rotates around itself. By integrating the angular velocity it is possible to obtain the angles of the device. However, due to drift this method does not give accurate results for a prolonged time. What is drift you wonder? Well if the device is standing still, the angular velocity should be exactly equal to zero. This is not the case however, it will be slightly different and this error will accumulate over time when integrating to angles.
- **3D Orientation:** The 3D orientation is shown by the 3D model in the middle. The orientation is computed by combining the sensor data from all three sensors together. By combining all sensors it is possible to overcome the limitations of each sensor separately. The 3D orientation can be expressed in Euler angles: yaw, pitch, roll or in quaternions. Quaternions are a mathematical representation using 4 numbers. In many situations, quaternions are preferred because they do not have singularity problems like the Euler angles.
- **Gravity vector (g):** If the pozyx device is standing still, the acceleration is exactly equal to the gravity. The gravity vector is shown by the black line and is always pointing down. Notice that even when moving the device (which introduces an additional acceleration) the gravity vector still points down. This is due to the fusion algorithm that can separate gravity from an arbitrary acceleration;
- **Linear acceleration in body coordinates (g):** The linear acceleration is the acceleration that remains after the gravity has been removed. When you hold the device horizontal, pointed forward, and shake it from left to right the circle will also move from left to right in the plot. However, if you rotate the device by 90 degrees and shake it again from left to right, the circle will now move in a different direction. This is because the linear acceleration is expressed in body coordinates, i.e., relative to the device. Note that all the above sensor data is expressed in body coordinates.
- **Linear acceleration in world coordinates (g):** Once the orientation of the device is known, it is possible to express the acceleration in world coordinates. By doing this, the rotation of the device no longer affects the linear acceleration in the plot.

3.4.4 The code explained

We'll now go over the code that's needed to retrieve all sensor data. Looking at the code's parameters, we can see that we can once again use a remote Pozyx for this. This means that you could for example attach a Pozyx to a ball, and watch the ball's spin directly on your screen as well with the Pozyx.

3.4.4.1 Imports and setup

We import the pypozyx library and the IMU interrupt flag from its bitmasks, and pythonosc so that we can send the sensor data to Processing over OSC. The addition of the default time library import allows us to measure the time between different measurements.

We initialize the PozyxSerial and UDP socket over which we'll send the data, and the setup then sets the last measured time right.

```

1  from time import time
2  from pypozyx import *
3  from pypozyx.definitions.bitmasks import POZYX_INT_MASK_IMU
4  from pythonosc.osc_message_builder import OscMessageBuilder
5  from pythonosc.udp_client import SimpleUDPClient

```

```

1  def setup(self):
2      """There is no specific setup functionality"""
3      self.current_time = time()

```

3.4.4.2 Loop

The main loop deserves to be elaborated on. The Pozyx's IMU sensors trigger an interrupt flag when there is new data available, and in the code we wait for this flag to trigger explicitly.

When checkForFlag returns POZYX_SUCCESS, meaning that POZYX_INT_MASK_IMU was raised and new IMU data is available, or when we're retrieving sensor data remotely, all sensor data and the calibration status will be read from the (remote) Pozyx and packed in a OSC message, which is then interpreted by the Processing sketch. You can see that there is a SensorData object available for this, which automatically converts the sensor data to its respective standard units.

```

1  def loop(self):
2      """Gets new IMU sensor data"""
3      sensor_data = SensorData()
4      calibration_status = SingleRegister()
5      if self.remote_id is not None or self.pozyx.checkForFlag(POZYX_INT_MASK_IMU, 0.01) ==
        POZYX_SUCCESS:
6          status = self.pozyx.getAllSensorData(sensor_data, self.remote_id)
7          status &= self.pozyx.getCalibrationStatus(calibration_status, self.remote_id)
8          if status == POZYX_SUCCESS:
9              self.publishSensorData(sensor_data, calibration_status)

```

The calibration status gives information about the quality of the sensor orientation data. When the system is not fully calibrated, the orientation estimation can be bad. The calibration status is a 8-bit variable where every 2 bits represent a different piece of calibration info.

3.4.5 What's next?

You're at the end of the standard tutorials that introduce you to all of Pozyx's basic functionality: ranging, positioning, and its IMU. If you haven't already done so and have the number of devices necessary, you can read the multitag tutorial. When you start work on your own prototype, don't be afraid to delve into our documentation which should suffice for all your needs.

3.5 Tutorial 4: Multitag positioning (Python)

3.5.1 Multitag positioning

The multitag positioning is an extension of tutorial 2, where we went over the positioning of a single device, and this being an extension is also visible in the code. Therefore, this tutorial will seem rather short and will go over the additions to allow for multitag. It is thus essential to follow that tutorial first.

For this tutorial you'll need at least four anchors and at least three other powered devices. These can be either tags or anchors, but remember that anchors don't have an IMU.

If all tools are installed, open up `multitag_positioning.py` in the Pozyx library's tutorial folder. Probably, the path to this file will be `"Downloads/Pozyx-Python-library/tutorials/multitag_positioning.py"`. You can run the script from either command line or a text editor that allows running the scripts as well. If you're on Windows and have installed Python, you might be able to simply double-click it. If you get an error, you likely forgot to install `pythonosc`, you can do this easily using `pip install python-osc`.

3.5.2 Plug and play

Looking at the parameters of the multitag positioning, you will see one major addition over the regular positioning example, namely the `tags` parameter which replaced the `remote_id` and `remote` parameters. In this list, you'll have to write the IDs of the remote devices you'll be using. For your anchors, use the setup you measured out in the positioning tutorial. You can also again choose the dimension and algorithm used for the positioning.

```
tags=[0x0001, 0x0002, 0x0003] # remote tags
```

Once you put the IDs of your tags here, you can start with multitag positioning. You can use the positioning Processing sketch straight from the bat as well if you set your `use_processing` to `true` and set your Processing serial and port correctly. However, reading the terminal output offers a more deterministic way of knowing whether you've got it working.

3.5.3 Code additions and changes

As this is an extension of the ready to localize tutorial, the code has been modified to support multiple remote devices through the added `tags` parameter. Printing both error and position now takes an additional parameter: the ID of the device we're working with.

3.5.3.1 Anchor configuration

Each device needs to be configured with the anchors to successfully perform positioning. The anchor configuration is performed on every device using `setAnchorsManual`, which iterates over every remote device, and feedback on this configuration is printed.

```

1 def setAnchorsManual(self):
2     """Adds the manually measured anchors to the Pozyx's device list one for one."""
3     for tag in self.tags:
4         status = self.pozyx.clearDevices(tag)
5         for anchor in self.anchors:
6             status &= self.pozyx.addDevice(anchor, tag)
7         if len(anchors) > 4:
8             status &= self.pozyx.setSelectionOfAnchors(POZYX_ANCHOR_SEL_AUTO, len(anchors))
9         self.printConfigurationResult(status, tag)

```

3.5.3.2 Positioning loop

Positioning is done in the same way as the anchor configuration: every tag gets its turn. If successful, the position is printed and otherwise an error message gets printed. This is effectively a TDMA approach, where the remote devices won't interfere with each other, unlike if you perform positioning on different devices at the same time, which will not work.

```

1 def loop(self):
2     """Performs positioning and prints the results."""
3     for tag in self.tags:
4         position = Coordinates()
5         status = self.pozyx.doPositioning(
6             position, self.dimension, self.height, self.algorithm, remote_id=tag)
7         if status == POZYX_SUCCESS:
8             self.printPublishPosition(position, tag)
9         else:
10            self.printPublishErrorCode("positioning", tag)

```

3.5.4 Caveats and closing remarks

- Positioning multiple devices at once is performed in a TDMA manner so that they don't interfere with each other. If your devices crash, try adding a small delay between the devices positioning to add robustness.
- Since you're using TDMA, you're splitting the update rate between the different devices. If you'd start with 24 Hz for a single device, you'd get at most 12 Hz per device if you're using two devices on the same UWB settings. 8 Hz if you're using three, and so on. Depending on your usecase, this could be important.
- Having more devices in a line-of-sight of both your master device and your anchors might be harder depending on your use case. Account for this to smartly set up your anchors and master.
- If your area gets larger and you want to maintain a good update rate, adding more anchors which will loosen the constraints on your UWB settings is recommended.

Now that you finished all the tutorials, you should be all set for any usecase implementation that Pozyx is currently capable of. If you have any questions, remarks, or problems with the tutorials, please mail to support@pozyx.io¹⁶

3.6 Changing the network ID (Python)

Ever since saving Pozyx registers to its flash memory was implemented, changing your device's ID has become possible. It's possible to do this both locally and remotely, and that's what will be discussed in this short article.

¹⁶ <mailto:support@pozyx.io>

Open up the Python script in the Pozyx library's useful folder. If you downloaded your library, you can likely find this at "Downloads/Pozyx-Python-library/useful/change_network_id.py"

3.6.1 The code explained

3.6.1.1 Parameters

```

1  new_id = 0x1000          # the new network id of the pozyx device, change as desired
2  remote = True            # whether to use the remote device
3  remote_id = 0x6000       # the remote ID

```

The parameters are what you'd expect for changing the ID: the `new_id` the device will take, and whether we're working with a remote device with ID `remote_id`. Change these to suit your preferences. In the parameters shown above, running the code would mean that a remote device with ID `0x6000` would have its ID changed to `0x1000`.

3.6.1.2 Changing the ID

Now, let's look at the code that effectively reconfigures a device's network ID

```

1  def set_new_id(pozyx, new_id, remote_id):
2      print("Setting the Pozyx ID to 0x%0.4x" % new_id)
3      pozyx.setNetworkId(new_id, remote_id)
4      if pozyx.saveConfiguration(POZYX_FLASH_REGS, [POZYX_NETWORK_ID], remote_id) == POZYX_SUCCESS:
5          print("Saving new ID successful! Resetting system...")
6          if pozyx.resetSystem(remote_id) == POZYX_SUCCESS:
7              print("Done")

```

```

1  void setup(){
2      // ...
3      // Pozyx and serial initialization
4      Pozyx.setNetworkId(new_id, remote_id);
5      uint8_t regs[1] = {POZYX_NETWORK_ID};
6      status = Pozyx.saveConfiguration(POZYX_FLASH_REGS, regs, 1, remote_id);
7      if(status == POZYX_SUCCESS){
8          Serial.println("Saving to flash was successful! Resetting system...");
9          Pozyx.resetSystem(remote_id);
10     }else{
11         Serial.println("Saving to flash was unsuccessful!");
12     }
13 }

```

We can see that there are three steps involved in configuring a device with a new network ID. As of writing, `setNetworkId` doesn't change the network ID the Pozyx is using, and the Pozyx needs to be reset after its new ID has been saved to its flash memory. So these three steps are:

- Setting the Pozyx's new ID using `setNetworkId`
- Saving the Pozyx network ID register. Note that while the ID is a `uint16_t` value, we only use the first register address to save it. We don't need to save two registers.

- If saving was successful, we reset the device. After the reset, the device uses the new ID. All the while, textual feedback is given to the user.

3.6.2 Finishing up

Changing the ID does not seem to have an immediate purpose except for if you have devices with a double ID, but changing the ID can be convenient for other reasons as well to make your setup more maintainable. Identifying anchors with a prefix, for example. Naming your anchors 0xA001 up to 0xA004 or more. This is easier to read than the default device IDs. If you have anchors on different UWB channel, naming the set on channel 5 0xA501 to 0xA504 and the set on channel 2 0xA201-0xA204 will make your life easier, and so on.

3.7 Troubleshoot basics (Python)

3.7.1 Overview

In this article, some basic code snippets are provided that can help you figure out any problems when things aren't going as they should. If you haven't read the getting started, please do so and install the necessary tools and libraries.

These code snippets can be found in a Python script as well, which can be used for troubleshooting a local and remote device straight from the bat. You can find this file in the Python library's useful folder. If you downloaded the library directly from GitHub, its path is probably "Downloads/Pozyx-Python-library/useful/basic_troubleshooting.py".

We take the code snippet approach in this article because being able to re-use these in your own projects is the ultimate goal here. Here is an overview of the troubleshooting basics:

- Device check
- Network check
- Interpreting the LEDs

3.7.2 Local and remote device check

The first thing to do to get an insight in your Pozyx's operation status, is reading out its basic status registers.

Registers can be read using the function `Pozyx.getRead()` for your local device or for a remote device. A full overview of all the registers with their description can be found [here](#). Let's read out the first 5 bytes starting from the memory address `POZYX_WHO_AM_I` and display the result.

```
1 pozyx.getRead(POZYX_WHO_AM_I, data, remote_id=remote_id)
2 print('who am i: 0x%0.2x' % data[0])
3 print('firmware version: 0x%0.2x' % data[1])
4 print('hardware version: 0x%0.2x' % data[2])
5 print('self test result: %s' % bin(data[3]))
6 print('error: 0x%0.2x' % data[4])
```

For a tag, the output should show the following result:

```

1  who am i: 0x43
2  firmware version: 0x10
3  hardware version: 0x23
4  self test result: 0b111111
5  error: 0

```

From these results you can already learn a great deal:

- **who am i** is wrong: the Pozyx device is not running or it is badly connected with the Arduino. Make sure that the jumper is on the BOOT0 pins.
- **firmware version** is wrong: you must update the firmware as explained in updating the firmware. Make sure that all devices, anchors and tags are running on the same firmware version.
- **Hardware version** is wrong: is your device on fire?
- **Self test** is different: for the tag the result should be 0b111111, for the anchor it should be 0b110000. Check out POZYX_ST_RESULT for more information. Note: in firmware version 1.0, the self-test might currently display 0b110000 instead of 0b111111. This is an anomaly which occurs when the selftest is done before the sensors are initialized. Don't panic, this doesn't necessarily mean the sensors aren't initialized. You can try the third tutorial to quickly see whether the IMU is actually operational. Check if the problem remains after resetting the device.
- **The error code is not 0:** Something went wrong, this isn't necessarily dramatic. Check out POZYX_ERRORCODE to see which error was triggered. If you see error 0x09, you can safely ignore this.

Note that the function `__init__` will check most of these registers as well to see if everything is working properly.

3.7.2.1 Not finding a remote device

When the `who am i` is not correct when reading out the register remotely, there may be additional causes:

- The remote address is wrong. It is printed on the label on the device.
- The devices are not within range or their signal is blocked for some reason.
- The devices are configured with different UWB settings. The UWB settings must be the same to enable communication. See POZYX_UWB_CHANNEL and the subsequent registers.
- Multiple Pozyx device are transmitting at the same time and are interfering with one-another.
- If you're using doDiscovery to find devices, as in the UWB configurator (Arduino), the slot duration might be too short for the UWB settings of the devices. This happens at channel 1, max preamble length.

3.7.3 Network check

For positioning, a number of tags and anchors are required. Despite the hardware differences, each device can operate in both modes, for positioning. Anchors won't have IMU data. This operation mode is selected with jumper on the T/A pins.

- **Tag mode:** the jumper is present. This device can move around and perform positioning
- **Anchor mode:** the jumper absent. This device must remain stationary and will support other devices in positioning.

With the function `Pozyx.doDiscovery()` it is possible to obtain a list of all the devices within range. The function takes one parameter to discover either tags, anchors or both. The constants for this parameter is POZYX_DISCOVERY_ALL_DEVICES for all devices, POZYX_DISCOVERY_TAGS_ONLY for finding only tags and, as you've likely guessed, POZYX_DISCOVERY_ANCHORS_ONLY for anchors. The following code will find and display all devices within range:

```
pozyx.clearDevices(remote_id)
if pozyx.doDiscovery(discovery_type=POZYX_DISCOVERY_ALL_DEVICES, remote_id=remote_id) == POZYX_SUCCESS:
    pozyx.printDeviceList(remote_id)
```

3.7.4 Interpreting the LEDs

The Pozyx device has a number of status LEDs that can also be used to analyze the Pozyx behavior at a glance.

- LED 1: This LED should be blinking slowly to indicate that the system is responsive.
- LED 2: This LED indicates that the Pozyx device is performing a specific function (for example, calibration or positioning).
- LED 3: This LED has no function.
- LED 4: This LED will indicate that there was an error somewhere. The error can be found by reading the register POZYX_ERRORCODE.
- RX LED: This indicates that the UWB receiver is enabled and that the device can receive wireless signals.
- TX LED: This LED indicates that the device has transmitted a wireless signal.

Using the register POZYX_CONFIG_LEDS the status LEDs can be configured to be turned off.