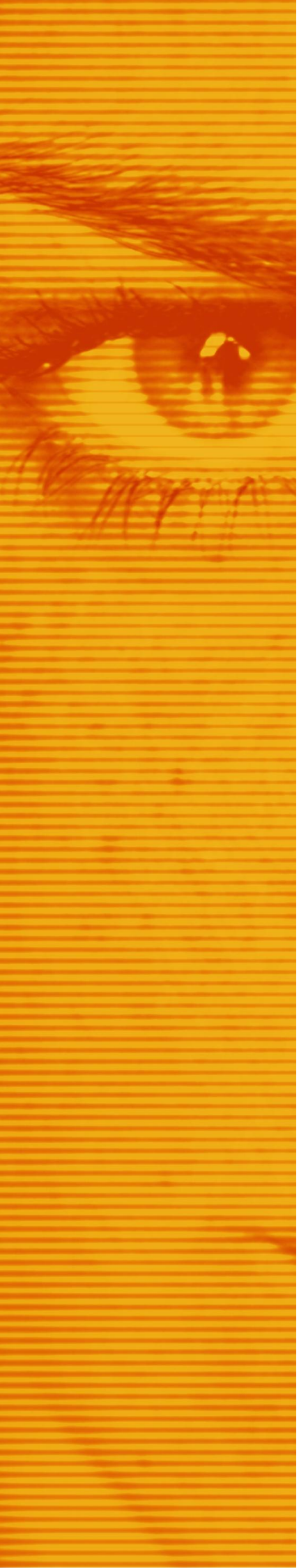


Tobii Analytics SDK

Developer's Guide

Release 3.0
May 8, 2013

This is the developer's guide for the Tobii Analytics Software Development Kit (SDK). It contains the documentation necessary to start developing eyetracking software for Tobii eyetracking hardware.

Table of content

Table of Figures and Tables

Tobii Analytics SDK

Introduction

This is the developer's guide for the Tobii Analytics Software Development Kit (SDK). It contains the documentation necessary to start developing eyetracking software for Tobii eyetracking hardware.

The core features of the Tobii Analytics SDK include:

- Browsing for and connecting to Tobii Eye Trackers.
- Calibrating the eye tracker to the user's eyes.
- Subscription to a stream of eyetracking data, including full access to and control over the high-resolution clock synchronization between client computer and eye tracker.

In addition to these core features, the SDK also includes functionality for:

- Querying and setting the calibration plane. This allows for finer integration of SDK applications and hardware as well as creating visualizations of the current setup.
- Real-time access to the track box to display accurate tracking status meters and correctly guide the user to a good position.
- Access to the firmware upgrade feature. This enables third-party applications to supply and perform firmware upgrades which then give them access to the most recently released features.
- Support for setting and getting eye tracker properties, such as setting the frame rate and requesting the coordinates of the current track box.

New features and enhancements

Changes between Tobii Analytics SDK version 2 and 3

The Tobii Analytics SDK went through a major update between versions 2 and 3. The two are totally different from a technical implementation perspective and client applications written for version 2 will need code changes to use version 3. However, authors of client applications should find that migration is fairly easy because the core concepts remain the same.

Changes from Tobii SDK version 2

The most important changes are:

- Access to the full spectrum of Tobii eyetracking hardware features
    - **More gaze data content such as 3D coordinates for eye position and gaze point.** This allows for easier integration in 3D environments and development of gaze angle-based eye movement filters.
    - **Possibility to both set and retrieve settings for the calibration plane from the eye tracker.** This enables finer integration of SDK applications and hardware as well as visualization of the current setup.
- **Full access to and control over the high-resolution clock synchronization between client computer and eye tracker.** This makes the synchronization procedure more transparent as compared to the previous implementation.
- **Access to the firmware upgrade functionality.** This enables third-party applications to perform firmware upgrades (which in turn allows for applications that depend on very recent features).

- **Available on several platforms.** The SDK version 3 is currently available on Windows, Linux, and Mac OS X.
- **Includes language bindings and code samples for many popular languages and frameworks: C++, C++/Qt, .NET, Python, and Matlab.**
- **Deployed on a per-application basis.** Several different versions of the SDK can be in use at the same time by different applications on the same computer and communicate with the same eye tracker.
- **No user interface and no GUI components.** The client application has responsibility for all interaction with the end user. However, several sample applications are available to use as a starting point when developing GUI components of eye tracking applications.

Changes from Tobii SDK 3.0.2 RC1

New features

- Matlab language binding.
- Python binding for Mac OS X.
- Support for synchronization signals from the Trig port on the Tobii TX300 Eye Tracker.

Documentation

- The Developer's guide has been restructured. Each language binding is now described in its own section.

Windows platform

- The SDK is now distributed as one zip archive file for 32-bit Windows and one for 64-bit Windows.
- The 32-bit SDK now includes both 32-bit and 64-bit installers for Bonjour, since the SDK can be used on both 32-bit and 64-bit Windows but the Bonjour bitness must match the operating system.

Linux platform

- The SDK is now distributed as a tar.gz archive file instead of as a Debian package.

Mac OS X platform

- The SDK is now distributed as a zip archive file instead of as a disk image file.
- The build process for C++ has changed and is more similar to Linux.
- The SDK no longer supports Objective-C/Cocoa.
- Python binding.

.NET Framework language binding

- Names of namespaces and classes have been changed to align better with .NET coding standards. This has mostly been a matter of casing, e.g., EyeTracker instead of Eyetracker.
- The asynchronous methods on the IEyeTracker interface have been changed to align better with .NET coding standards.
- The language binding assembly has been renamed to Tobii.EyeTracking.IO.dll.

Python language binding

- Module names and namespaces have been changed to harmonize with the other language bindings.
- Support for MAC OS X.

C++ language binding

- Two new code samples: one command-line application and one using the Qt GUI framework.
- It is no longer necessary to use a specific version of the Boost library.
- An issue with Visual Studio 2010 and stdint.h has been fixed. It is now possible to use the SDK with Visual Studio 2010 without modifications.

- The core library has been renamed from tobiisdk to tetio.
- Improved structure and naming conventions.

Objective-C/Cocoa language binding

- The SDK no longer supports Objective-C/Cocoa.

System requirements

Supported platforms, languages, and frameworks

The SDK supports the following platforms, languages, and frameworks:

Platform	Windows 32 bit	Windows 64 bit	Linux 32 bit	Linux 64 bit	Mac OS X 64 bit
Versions	*8, 7, Vista*, XP**	*8, 7, Vista*, XP**	*Ubuntu 12.04 LTS*	*Ubuntu 12.04 LTS*	*10.6 (Snow Leopard) and later*
.NET Framework, version 2.0 and later	Yes	Yes	Yes (Mono)*	Yes (Mono)*	Yes (Mono)*
Python 2.7	Yes	Yes	Yes	Yes	Yes
C++	Yes	Yes	Yes	Yes	Yes
Matlab 2012	Yes	Yes	No	No	Yes

**Should work, but not officially supported*

Supported eye trackers

The following Tobii Eye Trackers are supported: **Tobii X60, Tobii X120, Tobii T60**, **Tobii T120, Tobii T60 XL, Tobii TX300**, **Tobii X1 Light, Tobii X2-30 and Tobii X2-60**.

Models that are not supported are: **Tobii 1750, Tobii X50, Tobii 2150, Tobii 1740 (D10), Tobii P10, C-Eye**, **PCEye**, **Tobii IS-1 and Tobii Rex.**

Supported firmware versions

To use the Tobii SDK 3.0 with any of the supported Tobii Eye Tracker models, the eye tracker firmware needs to be compatible. A firmware upgrade might be required. To see how to determine the firmware version and upgrade the firmware using the Eye tracker browser application, please refer to *Appendix 1. The Eye tracker browser and firmware upgrade.* In general, always make sure to use the *latest available* firmware for your eye tracker model.

Firmware versions that support the Tobii SDK 3.0 are:

- Tobii X60, X120, T60 and T120: **2.0.0** *(or higher)*
- Tobii TX60 XL: **2.0.0** *(or higher)*
- Tobii TX300: **1.0.0** *(or higher)*
- Tobii X1 Light: **1.4.0** *(or higher)*
- Tobii X2 Series: **All FW versions**

NOTE: A firmware upgrade in order to support the Tobii SDK 3 will never break compatibility with older SDKs. Applications written using older SDKs will keep working with all above listed models, and applications using the new SDK and applications using the old SDK can run side-by-side without problems.

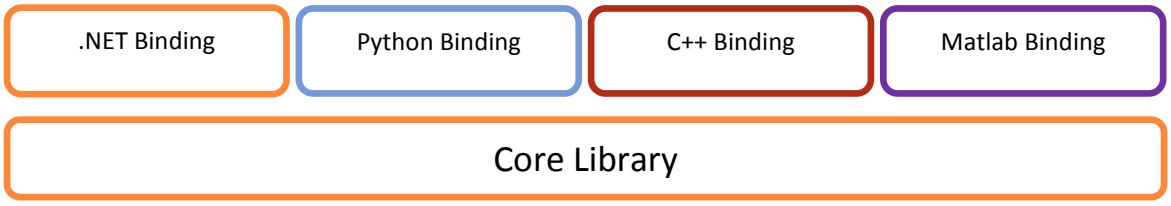
Getting Started

The recommended way to get started with application development using the Tobii Analytics SDK is to start by reading this section and the section *Developing applications using the Tobii Analytics SDK* in this manual, then the section about your language binding of choice. The next suggested step is to try the code samples and browse through the source code to get an idea of how everything fits together. It is, of course also possible to start with the sample code and work backwards from there.

Language bindings

The Tobii Analytics application programming interface (API) is implemented as a core library, "tetio", and a set of language bindings which are built on top of the core. The purpose of the language binding layer is to expose the core functionality in a way that is typical for the language and framework – in other words, to make the API feel natural to developers familiar with those languages.

Consequently, the first choice that the developer faces is which language binding to use.



Installing the SDK on Windows

Prerequisites

The Tobii Analytics SDK for Windows uses Bonjour for locating eye trackers on the local network and connected using USB. Bonjour is an implementation of the Zero configuration network technology and is available free of charge from Apple Computer, Inc. The Bonjour installers for 32-bit and 64-bit Windows are also included with the Tobii Analytics SDK for Windows.

NOTE: The bitness of Bonjour that you install must match the operating system. It is possible to install 32-bit Bonjour on a 64-bit Windows system, but then it will only work with 32-bit client applications.

The core library depends on the Microsoft C Run-Time Libraries, version 90, and will not function properly unless these libraries are installed on the computer. The run-time libraries can be downloaded from Microsoft, free of charge, using these links:

Microsoft Visual C++ 2008 SP1 Redistributable Package (x86)
http://www.microsoft.com/en-us/download/details.aspx?id=5582

Microsoft Visual C++ 2008 SP1 Redistributable Package (x64)
http://www.microsoft.com/en-us/download/details.aspx?id=2092

Installing the Tobii Analytics SDK

Download the appropriate zip archive file from tobii.com and unpack it to a directory of choice, e.g., "C:\tobiisdk". The choice of zip archive file depends on the bitness of the application that you intend to develop. This means that you might actually need both packages:

- tobii-analytics-sdk-<version>-win-Win32.zip for 32-bit application development.
- tobii-analytics-sdk-<version>-win-x64.zip for 64-bit application development.

Browse the directory where you unpacked the SDK to find the following directories and files:

Directory/File	Description
Bonjour	Bonjour installers
Cpp	C++ language binding and code samples
DotNet	.NET language binding and code samples
Eye tracker browser	The Eye tracker browser application
Matlab	Matlab language binding and code sample
Python27	Python language binding and code samples
Tobii Analytics SDK Developer Guide.pdf	Developer's guide
Tobii Analytics SDK License agreement.pdf	License agreement

Installing the SDK on Linux

NOTE: This release of the Tobii Analytics SDK targets the Ubuntu 12.04 LTS distribution of Linux. It may work with other versions and distributions as well, but this is the version it was designed and tested for.

Prerequisites

The Tobii Analytics SDK for Linux uses Avahi for locating eye trackers on the local network and connected using USB. Avahi basically provides the same functionality as Bonjour does on Windows.

The core library depends on the following components, which means that they must be present in order to run any client applications:

- libssh2-1
- libavahi-client3

The following components may also need to be installed, depending on the intended usage:

- libqtgui4, in order to run the Eye tracker browser and the C++/Qt sample application.
- qt4-dev-tools, in order to develop and build client applications using C++/Qt.
- python-dev, in order to run any Python applications.
- boost – downloaded, built, and installed as described in the README-boost.txt file.

NOTE: On some Ubuntu machines Avahi does not start when it detects a unicast DNS **.local** domain. However, sometimes this check for unicast DNS .local domains appears to give false-positives. In these cases a workaround is required in order to disable the check[1]. In order to disable the check for unicast .local domain, create the file: `/etc/default/avahi-daemon` with the contents `AVAHI_DAEMON_DETECT_LOCAL=0`, and then restart the avahi-daemon.

Installing the Tobii Analytics SDK

Download the appropriate tar.gz archive file from tobii.com and unpack it to a directory of choice, e.g., "~/tobiisdk". The choice of archive file depends on the bitness of the operating system:

- tobii-analytics-sdk-<version>-ubuntu-i386.tar.gz for 32-bit Linux distributions.
- tobii-analytics-sdk-<version>-ubuntu-amd64.tar.gz for 64-bit Linux distributions.

[1] https://bugs.launchpad.net/ubuntu/+source/avahi/+bug/327362

Browse the directory where you unpacked the SDK to find the following directories and files:

Directory/File	Description
bin	Executable files: currently only the Eye tracker browser
cpp	C++ language binding and code samples
python27	Python language binding and code samples
Tobii Analytics SDK Developer Guide.pdf	Developer's guide
Tobii Analytics SDK License agreement.pdf	License agreement

Installing the SDK on Mac OS X

Download the file tobii-analytics-sdk-<version>-osx-x86_64.zip from tobii.com and unpack it to a directory of your choice, e.g., "~/tobiisdk".

Browse the directory where you unpacked the SDK to find the following directories and files:

Directory/File	Description
Applications	Executable files: currently only the Eye tracker browser
cpp	C++ language binding and code samples
Matlab	Matlab language binding and code sample
Python27	Python language binding and code samples
Tobii Analytics SDK Developer Guide.pdf	Developer's guide
Tobii Analytics SDK License agreement.pdf	License agreement

Developing applications using the Tobii Analytics SDK

A common usage scenario of the Tobii Analytics SDK involves the following steps:

- browsing for available eye trackers,
- connecting to an eye tracker,
- (optionally) configuring the eye tracker (calibration plane, frame rate, etc.),
- (optionally) calibrating the eye tracker,
- (optionally) synchronizing clocks with the eye tracker,
- collecting eye gaze data and possibly acting on it in real time, e.g., for gaze-contingency studies.

This section will describe those tasks, plus a few less common ones, in generic terms but without going into implementation details. Please refer to the section describing your language binding of choice for a description of the actual API.

Coordinate systems used by Tobii Eye Trackers

When using the Tobii Analytics SDK, the eye position and gaze data is available in either one or more of the coordinate systems.

User Coordinate System (UCS)

All the data available from Tobii Eye Trackers that describe **3D space coordinates** are given in the so-called *User Coordinate System* or *UCS* for short. The UCS is a millimeter-based system with its origin at the centre of the frontal surface of the eye tracker (see Figure 1). The coordinate axes are oriented as follows: the x-axis points horizontally towards the user's right, the y-axis points vertically towards the user's up and the z-axis points towards the user, perpendicular to the front surface of the eye tracker.

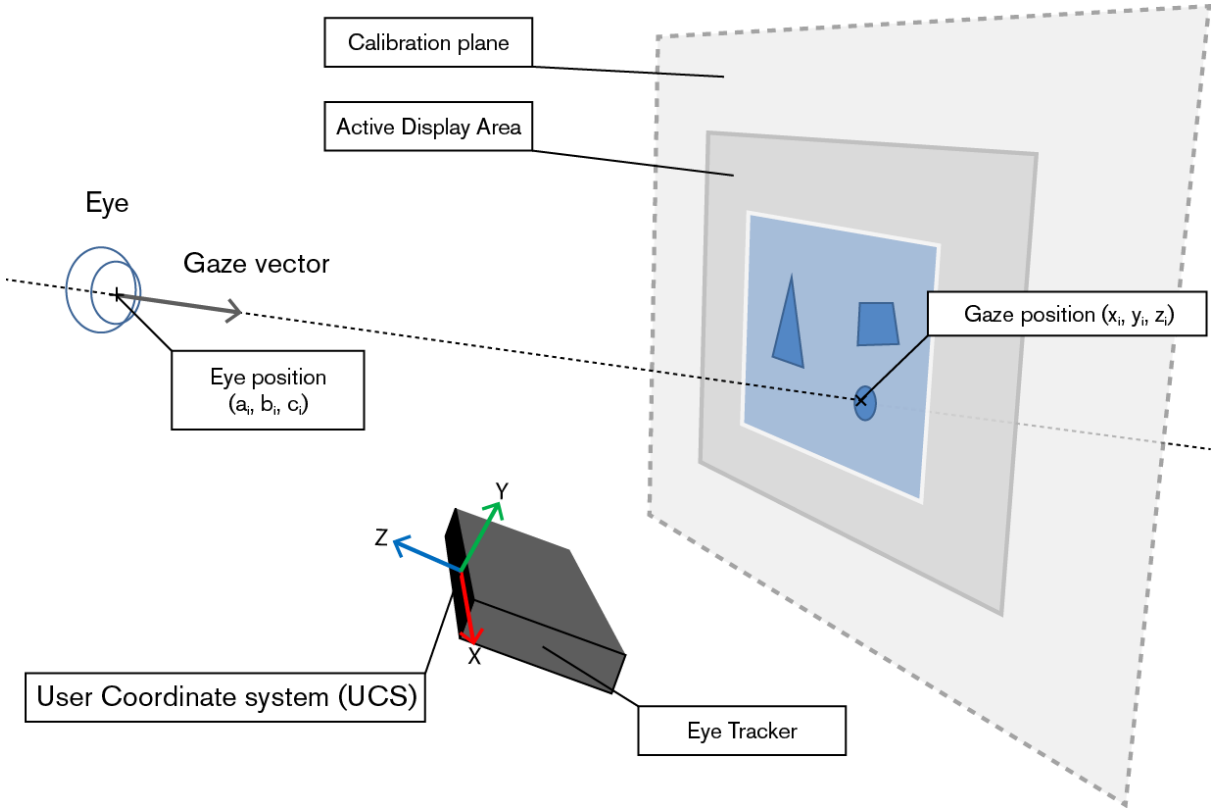


Figure 1. User Coordinate System (UCS)

Active Display Coordinate System (ADCS)

All gaze data is also mapped into a **2D coordinate system** aligned with the Active Display Area. When using an eye tracker with a monitor, the Active Display Area is the display area excluding the monitor frame. For X-series eye trackers, the Active Display Area is the area used to show the calibration points when doing the calibration of the eye tracker to the user's eyes. The origin of the *Active Display Coordinate System* (ADCS) is at the upper left corner of the Active Display Area. The point (0, 0) denotes the upper left corner and (1, 1) the lower right corner of the Active Display Area (see Figure 2).

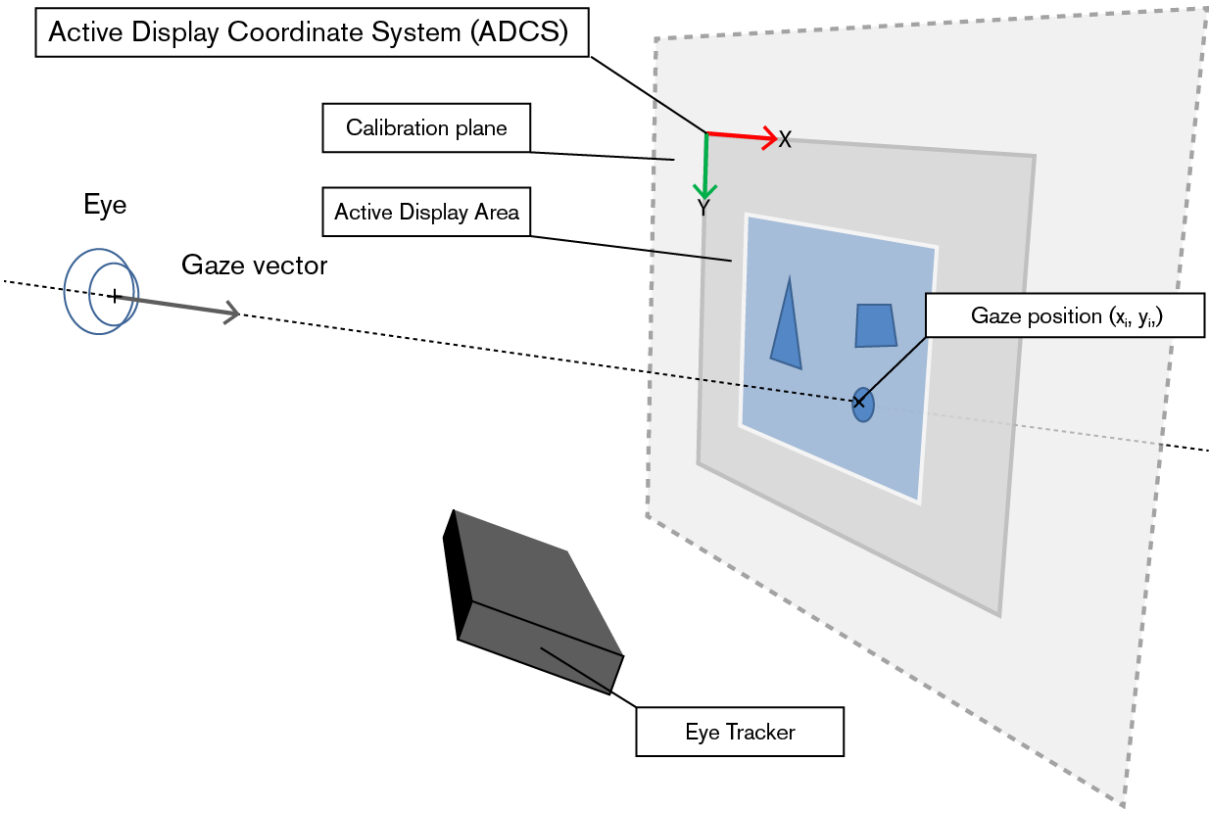


Figure 2. Active Display Coordinate System (ADCS)

NOTE: Eye tracker units that are not fixed to a screen, e.g., the Tobii X60 and Tobii X1 Light, must be configured to use the proper calibration plane and Active Display Area before use.

Track Box Coordinate System (TBCS)

The track box is the volume in which the eye tracker is theoretically able to track the eyes. Thus, the subject is able to move the head freely and still remain trackable as long as the eyes are still within the box. In the SDK there are methods to find out the size and position of the track box volume, but also where the eyes are within it. The coordinate system used to describe the position of the eyes within the track box is called the *Track Box Coordinate System (TBCS)* (see Figure 3). The TBCS has its origin in the top, right corner, which is located closest to the eye tracker (0, 0, 0). The TBCS is a normalized coordinate system, which in this case means that the max value for all coordinates is (1, 1, 1) and the location of that point is the lower, left corner furthest away from the eye tracker.

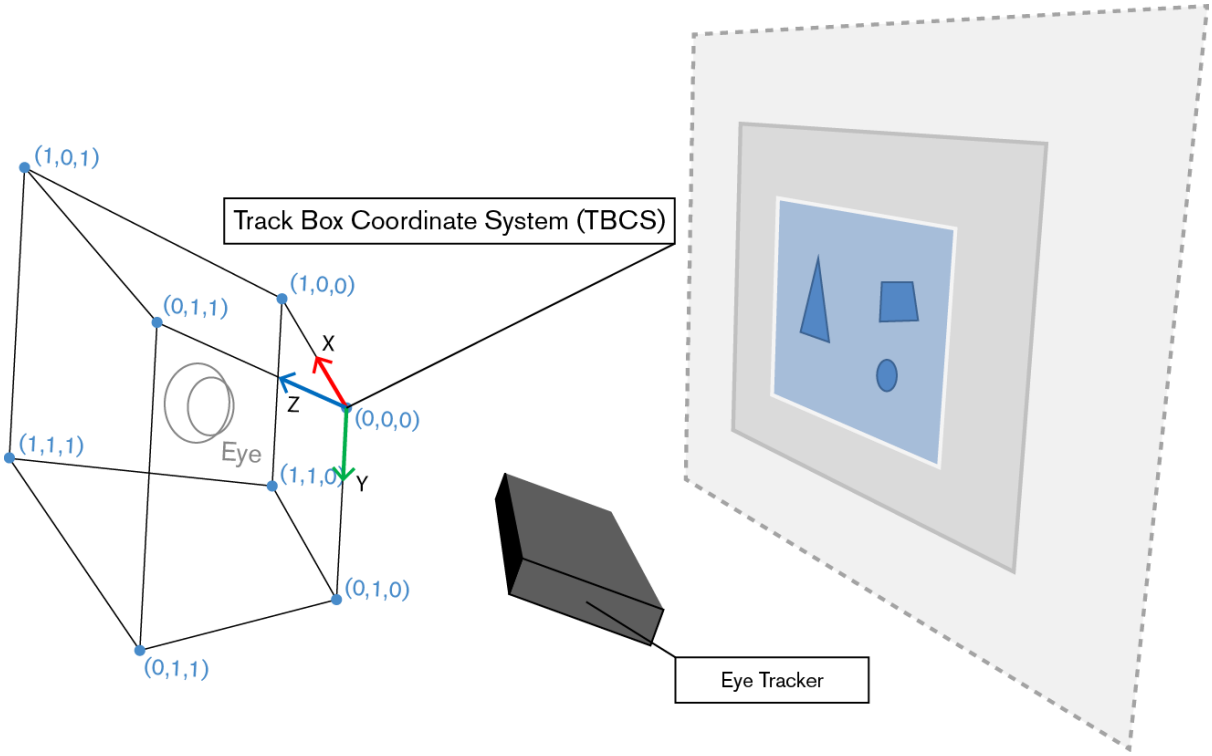


Figure 3. Track Box Coordinate System.

Browsing for available eye trackers

All Tobii Eye Trackers present themselves on the local area network using a technology called Zero configuration networking, or *Zeroconf*[2] for short.

NOTE: The Tobii X1 Light Eye Tracker (which is connected to its host computer using USB) actually creates a kind of mini network between itself and the host computer when connected, and therefore it will also be reported as present on the network even though no other computers can reach it.

To browse for available eye trackers, create an instance of the EyeTrackerBrowser class and subscribe to its events[3]. Then call its start method. The EyeTrackerBrowser class will notify its subscribers when:

- An eye tracker appears on the network.
- An eye tracker disappears from the network.
- An eye tracker has one of its properties changed.

The events (except "disappear") also carry some metadata about the eye tracker in the form of a set of properties. These are summarized in Table 1.

When the user has selected an eye tracker, or when you have found the eye tracker you were looking for (in case you already had selected one and stored its name for later reference), call the stop method to stop browsing. A common practice is to use the first eye tracker that is reported. However, this is only advisable if you know that there will only be a single eye tracker on the network.

[2] Zeroconf is a set of protocols that allows entities on a local area network to publish and discover services such as printers or eye trackers without the need to do any specific configuration.
[3] An event is a mechanism that allows an object to inform other objects when something interesting happens. Some of the supported languages, such as the .NET framework, have events as an intrinsic feature, whereas others do not. The C++ language wrapper uses boost signals, which is a generalization of the well known Observer design pattern.

Table 1. Eye Tracker Info Properties

Property	Description
Generation	A string describing the product generation. The currently existing values are “TX” (Tobii T60, Tobii T120, Tobii X60, Tobii X120), “XL” (Tobii T60 XL), “IS” (Tobii X1 Light) and “MSP” (TX300), but more will be added in the future.
Model	A string describing the product model, e.g. “Tobii T60”.
Version	A string that represents the firmware version of the specific unit. Usually on the format “1.2.3”, but other variations exist.
Product ID	A string that describes the product ID, e.g. “TT060-203-52300012”. This property is guaranteed to be unique among all Tobii products and can be used to identify a specific unit.
Status	A status string that describes the current status of a specific unit. Can have the following values: • “ok” – unit is working as expected • “not-working” – unit is up and running but the eye tracker firmware is not working as expected • “upgrading” – unit is installing a firmware upgrade • “error” – indicates a serious error. The unit is malfunctioning in some way
Given Name	A string describing the unit name. The name property can be changed by any SDK client application and is of no vital importance for the operations of the eye tracker.

The *Status* property is important since an eye tracker will only accept connecting client if it is in the “ok” state. If either “not-working” or “error” is received, try restarting the eye tracker. If the same error code is still received, reinstall the firmware or update to a newer version if such a version is available. If the problem persists, contact Tobii support.

Connecting to an eye tracker

After locating an eye tracker on the local network, the next step is to establish a connection. The connection is represented by the EyeTracker class. There are no connect or disconnect methods; the connection is held open for the lifetime of the EyeTracker object.

The EyeTracker class provides an event that is raised when, e.g., the network connection is lost. When this event has been received, the connection should be considered broken and no attempts should be made to further use the instance.

Collecting eye gaze data

The EyeTracker class provides an event that is raised whenever a gaze data packet is received from the eye tracker. Collecting eye gaze data is a two-step process: first, subscribe to the event, and then call the startTracking method. This will instruct the eye tracker to start delivering gaze data packets at its configured frame rate. Once started, the eye tracker will keep tracking and delivering gaze data until stopTracking is called, or until its last client disconnects. Note that gaze data packets are delivered regardless of whether the eye tracker was successful in tracking any eyes or not. The validity code (see the section *Validity code*) can be used to filter out invalid data packets.

A gaze data packet contains the following data[4]:

[4] See also the data reference in *Appendix 3. Tobii T, X, and TX eye tracker data reference*.

Timestamp

This is a value that indicates the time when the information used to produce the gaze data packet was sampled by the eye tracker. The value should be interpreted as a time interval in microseconds measured from an arbitrary point in time. The source of this timestamp is the internal clock in the eye tracker hardware. A typical use case is to synchronize the gaze data stream with some other kind of data stream, e.g., information about when a certain stimulus was presented on a computer screen. How this is done is described in the section called *Synchronizing clocks with the eye tracker* in this document.

Eye position

The eye position is provided for the left and right eye individually and describes the position of the eyeball in 3D space. Three floating point values are used to describe the x, y and z coordinate respectively. The position is described in the UCS coordinate system.

Relative eye position

The relative eye position is provided for the left and right eye individually and gives the relative position of the eyeball in the track box volume as three normalized coordinates. This data can be used to visualize the position of the eyes, which is commonly used as a tool to help the user position himself/herself in front of the tracker.

3D gaze point

The 3D gaze point (or gaze position) is provided for the left and right eye individually and describes the position of the intersection between the calibration plane and the line originating from the eye position point with the same direction as the gaze vector. The coordinates for the eye position and the gaze position are given in the User Coordinate System coordinate system. This is illustrated in Figure 4.

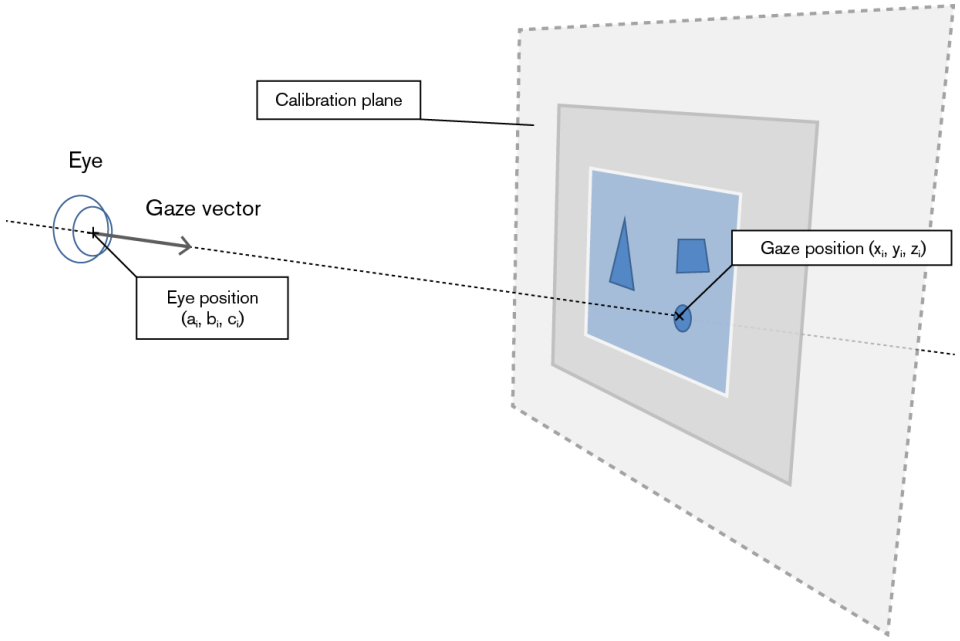


Figure 4. Illustration of the 3D gaze point and eye position.

The gaze vector can be computed by subtracting the 3D gaze point from the 3D eye position and normalizing the resulting vector.

2D gaze point

The 2D gaze point is provided for the left and right eye individually. It is conceptually the same as the 3D gaze point, but expressed as a two-dimensional point on the calibration plane instead of as a point in 3D space. The position is given in the Active Display Coordinate System where (0, 0) denotes the upper left corner and (1, 1) the lower right corner of the Active Display area.

Validity code

The validity code is an estimate of how certain the eye tracker is that the data given for an eye really originates from that eye. The validity code scale starts at 0, which signifies eye definitely found, and stops at 4, which signifies eye not found. When the tracker finds two eyes in the camera image, identifying which one is the left and which one is the right eye is straightforward. The same is true when no eyes are found at all. The most challenging case is when the tracker only finds one eye in the camera image. When that happens, the image processing algorithms try to deduce if the eye in question is the left or the right one. This is done by referring to previous eye positions, the position in the camera sensor, and certain image features. The validity codes describe the outcome of this deduction. The validity codes can only appear in certain combinations. These combinations and their interpretations are summarized in Table 2.

Table 2. Data Validity Codes

		Right Eye Validity Code				
Left Eye Validity Code		0	1	2	3	4
	0	Found two eyes	N/A	N/A	N/A	Found one eye. Most probably the left eye.
	1	N/A	N/A	N/A	Found one eye. Probably the left eye.	N/A
	2	N/A	N/A	Found one eye. The eye tracker cannot with any certainty determine which eye it is.	N/A	N/A
	3	N/A	Found one eye. Probably the right eye.	N/A	N/A	N/A
	4	Found one eye. Most probably the right eye.	N/A	N/A	N/A	No eyes found

The validity codes can be used to filter out data that is most likely incorrect. Normally it is recommended that all samples with validity code 2 or higher be removed or ignored.

Pupil Diameter

The pupil diameter data is provided for the left and the right eye individually and is an estimate of the pupil size in millimeters. The Tobii Eye Trackers try to measure the true pupil size, i.e. the algorithms take into account the magnification effect given by the spherical cornea as well as the distance to the eye.

Configuring the Active Display Area for an eye tracker ("X Config")

An eye tracker needs to know where the screen is relative to itself in order to produce meaningful gaze point data. Eye trackers that are not built into display monitors (e.g. the Tobii X-series eye trackers) must be configured for use with a particular display monitor, projector, or other planar surface.

The configuration consists of three points in space corresponding to the top left, bottom left, and top right corners of the Active Display Area, i.e. the area where stimuli will be shown and where the calibration points will be displayed during the calibration of the eye tracker to the user's eyes. If a monitor is used, the coordinates used are the locations of the corner pixels of the display area. The three points are to be given in the User Coordinate System (UCS) illustrated by the colored vectors in Figure 5. The UCS is a millimeter-based coordinate system with its origin in the middle of the front surface of the eye tracker. The x axis points to the user's right, the y axis points in the upward direction and the z axis points towards the user (see the section *User Coordinate System (UCS)*). Note that this coordinate system is eye tracker fixed, which means that tilting or moving the eye tracker invalidates the Active Display Area configuration.

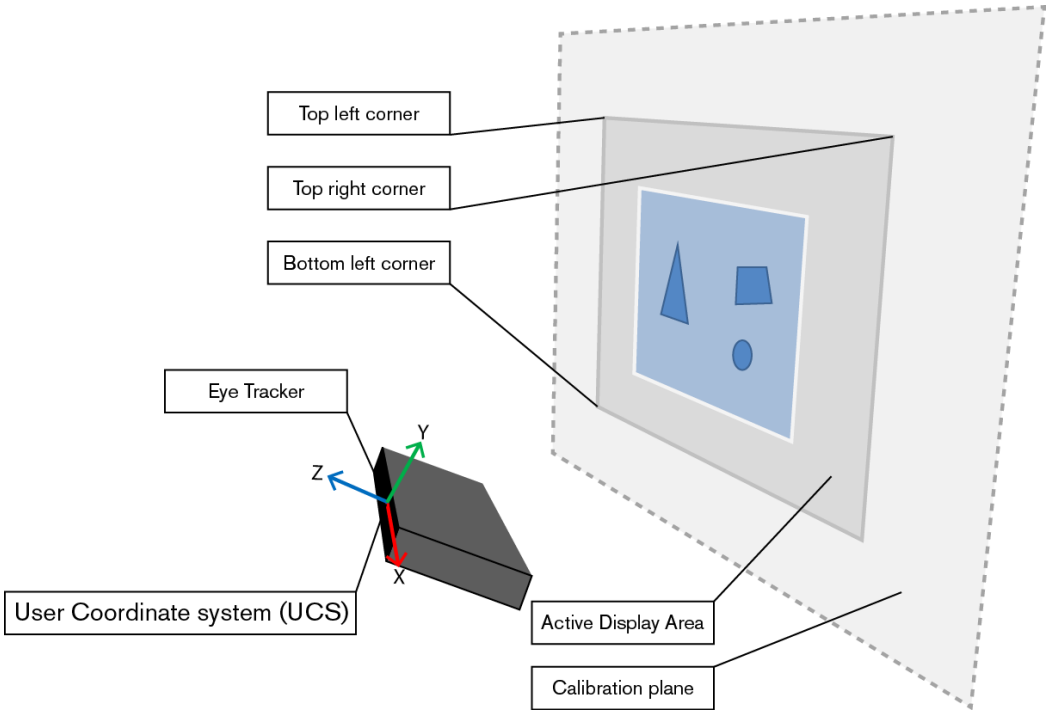


Figure 5. Eye tracker configuration schematic.

NOTE: It is important that the provided points correspond to the two sides of a rectangle. If the correct coordinates are not provided to the eye tracker, this will result in offsets in the gaze data.

It is also possible to query the eye tracker for its current Active Display Area configuration. This can be done for all eye trackers - not only X-series eye trackers.

NOTE: The eye tracker writes the Active Display Area configuration to persistent storage. It will be in use until a new Active Display Area configuration is written, regardless of whether the eye tracker loses power or is disconnected from the client application.

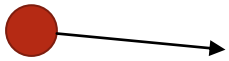
Calibrating an eye tracker

In order to compute the gaze point and gaze direction with high accuracy, the eye tracker firmware adapts the algorithms to the person sitting in front of the tracker. This adaptation is done during the calibration process when the user is looking at points located at known coordinates on the Active Display Area. The calibration is initiated and controlled by the client application.

NOTE: The calibration is reset when the eye tracker loses power or when a USB connected eye tracker loses its connection to the client computer.

Calibration procedure

1. The calibration of the eye tracker would typically be done as follows: A small animated object should be shown on the screen to catch the user's attention.



2. When it arrives at the calibration point, let the point rest for about 0.5 seconds to give the user a chance to focus. Shrink the object to focus the gaze.

3. When shrunk, tell the eye tracker to start collecting data for the specific calibration point.
4. Wait for the eye tracker to finish calibration data collection on the current position.
5. Enlarge the object again.
6. Repeat steps 1-5 for all desired calibration points.

The animation in step 1 should not be too fast, nor should the shrinking in step 2 be too fast. Otherwise the user may not be able to get a good calibration result due to the fact that he or she has no time to focus the gaze on the target before the eye tracker starts collecting calibration data.

The normal number of calibration points is 2, 5, or 9; more points can be used but the calibration quality will not increase significantly for more than 9 points. Usually 5 points yields a very good result and is not experienced as too intrusive by the user.

NOTE: It is possible to do a calibration without using a screen. The procedure is similar to what is described above, but instead of showing the points on a screen the points should be presented at the correct locations within the Active Display Area. This is typically done by drawing the points on a piece of cardboard with the equivalent size of the Active Display Area and positioned in the plane provided to the eye tracker when configuring the position Active Display Area. The user should then be asked to look at the points in a predefined order and prompted to switch focus from one calibration point to the next.

Configuring the calibration points

The location of the calibration points is decided by the client application. A typical calibration pattern for 5 points can be seen in Figure 6.

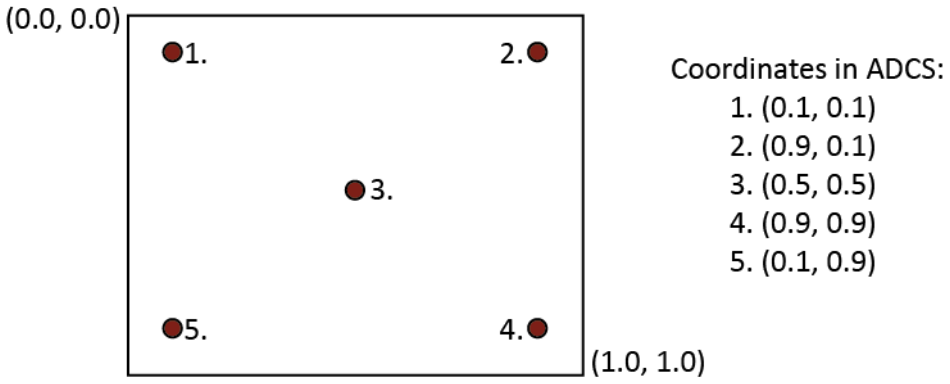


Figure 6. Standard 5-point Calibration Pattern.

NOTE: All points are given in normalized coordinates in such a way that (0.0, 0.0) corresponds to the upper left corner and (1.0, 1.0) corresponds to the lower right corner of the Active Display Area. When choosing the calibration points it is important to consider the following:

- The calibration points should span an area that is as large or larger than the area where the gaze controlled application or the stimuli is to be shown in order to ensure good interaction.
- The calibration points must be positioned within the area that is trackable by the eye tracker and be within the Active Display Area.

The calibration state

To be able to perform a calibration the client application must first enter the calibration state. The calibration state is an exclusive state which can only be held by one client at a time. It is entered by calling the method StartCalibration() and is left by calling StopCalibration(). Whenever a client enters or leaves the calibration state, the notifications CalibrationStarted and CalibrationStopped are sent to all clients connected to the same eye tracker except the calibrating client.

Some operations can only be performed during the calibration state, e.g. AddCalibrationPoint(), RemoveCalibrationPoint(), ClearCalibration() and ComputeCalibration().

Other operations such as SetCalibration() or GetCalibration() work at any time. However, SetCalibration() can only be called by the client, which is currently in the calibration state. These notifications are mostly meant for user interface purposes, like graying out a “Calibrate” button etc. Since the communication with the eye tracker is asynchronous, best practice is to call StartCalibration() and check whether it reports that another client is currently calibrating rather than caching the result of the calibration events .

Calibration buffers

The eye tracker firmware uses two buffers to keep track of the calibration data:

- The temporary calibration buffer. This buffer is used only during calibration. This is where data is added or removed.
- The active calibration buffer. This buffer contains the data for the calibration that is currently set.

The active calibration buffer is modified either by a call to SetCalibration(), which copies data from the client, or a successful call to ComputeCalibration, which computes calibration parameters based on the data in the temporary buffer and then copies the data into the active buffer.

There is also an API function to download the active calibration from the eye tracker to the client application for applications that wish to store a calibration for later use.

Calibration plots

If you have previous experiences with any of Tobii’s eyetracking products it is very likely that you have seen the calibration plot which illustrates the calibration results. The calibration plot is a simple yet concise representation of a performed calibration and it usually looks something like what is shown in Figure 7. However, the presentation design can vary.

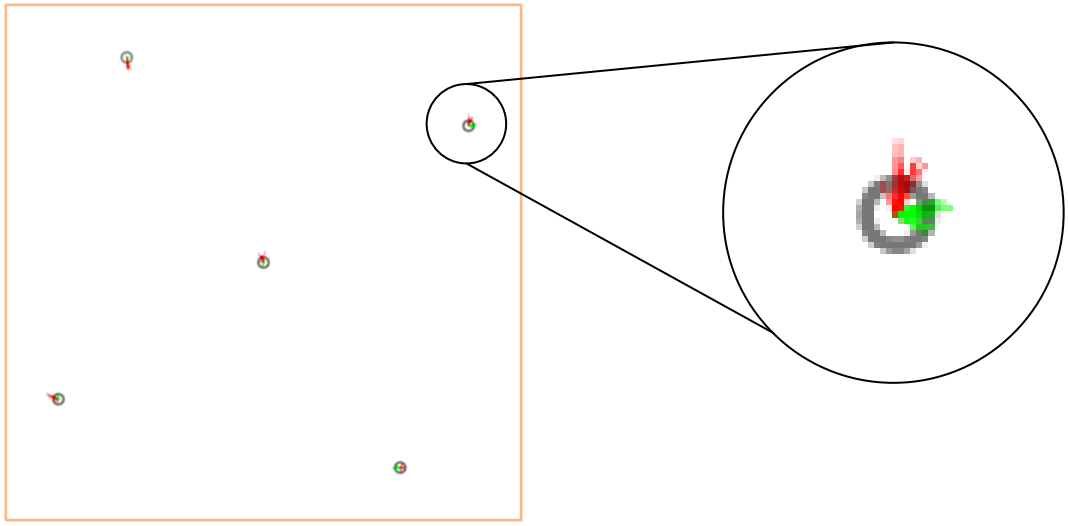


Figure 7. 5-point calibration plot with combined data for both the left and the right eye

The calibration plot shows the offset between the mapped gaze samples and the calibration points based on the best possible adaptation of the eye model to the collected values done by the eye tracker during calibration. In Figure 7, the samples are drawn as red and green where the lines represent the offset between the sample points and the calibration points. The red lines represent samples from the left eye and the green lines represent samples from the right eye. The grey circles are the actual calibration points. The data displayed in the plot is made available to client applications through the language bindings. This allows for alternative visualizations of calibration results as well as the traditional visualization as seen in Figure 7.

Synchronizing clocks with an eye tracker

Getting the current time

When writing an application that uses gaze data, it is often necessary to merge the data with data from other sources. This in turn means that you have to compare the data using a common time base. With that in mind, some basic clock functionality has been included in the SDK. The time functionality is exposed through the Clock class that is available for all supported languages and for all the different platforms. A Clock instance is a very simple object that can do two things:

- Provide the current time on the client computer. The time is given as a microsecond value stored in a 64 bit signed integer. The base time (t=0) is unspecified and can differ between different operating systems depending on the properties of the underlying hardware clock.
- Provide the clock resolution. This is a property of the clock and doesn't change over time. The resolution is also given as a microsecond value stored in a 64 bit signed integer.

Synchronizing clocks with the eye tracker

In order to correctly correlate stimulus events with gaze data, the high-resolution clock on the eye tracker must be synchronized with the high-resolution clock on the client computer. In previous versions of the SDK, this synchronization process has been completely hidden from the user. In this version, we have taken a more transparent approach. This allows for a better understanding of how the time synchronization works and also makes it possible to handle errors that might occur in the synchronization.

The gaze data reported by the eye tracker is time-stamped with a time from the high-resolution clock built into the eye tracker hardware. This clock is relative with no known start time. In the remainder of this document we will refer to this clock as the "eye tracker clock".

Usually, the timestamps from the clock on the client computer are of the same kind, i.e. relative and without a known starting point. We refer to this clock as the “SDK clock”. The SDK clock can be a built-in SDK clock, which uses QueryPerformanceCounter() on Windows and clock_gettime() on POSIX. It is also possible to create a user-defined clock (for example an NTP clock) and use it with the SDK as the SDK clock.

If the two clocks were to tick at exactly the same rate, it would be enough to compute the offset between them to get a decent synchronization. However, since the two clocks are based on computer hardware and no computer clocks are perfect, the two clocks will most likely drift in relation to each other. There are many reasons for this: One is that the clocks on a typical motherboard can have different tick rates due to imperfections in the clock crystals. Another reason is that the tick rate typically depends on external factors such as temperature. This means that the tick rate of a clock can change over time. To account for this, the synchronization algorithm in the SDK always bases the synchronization on at least two measurements performed at different times. This gives us the possibility to not only estimate the clock offset, but also to estimate the drift of the two clocks.

The synchronization process is managed by a synchronization manager class. This class is responsible for synchronizing a local clock (e.g. the SDK clock) with the high-resolution clock in the eye tracker. This class has two methods; convert_from_local_to_remote() and convert_from_remote_to_local(), which can perform conversion from remote to local (i.e. client computer) timestamps and vice versa. These conversions are guaranteed to be performed offline, and therefore do not involve network traffic to the eye tracker. A description of the synchronization process can be found in the section *Synchronization process.*

The SynchronizationManager also has two events or callbacks: an error handler for asynchronous error reporting and a status change handler, which will be invoked when the synchronization status is updated or when a synchronization run has been performed.

The current synchronization status of a SynchronizationManager is described by a synchronization status flag, which can have the following values:

- Unsynchronized:
  - The two clocks are not synchronized.
- Stabilizing:
  - Synchronization has started but only uses one sync point. This compensates for clock offset but not clock drift over time (see the section *Synchronization process*).
- Synchronized:
  - The two clocks are synchronized using a two-point synchronization that takes into account both clock offset and clock drift (see the section *Synchronization process*).

There are two ways to get the synchronization status: either you can send a query to the synchronization manager and ask for its current status, or you can listen for the event containing the current status which is sent from the synchronization manager. The synchronization status contains the following information:

- Synchronization state flag as previously described.
- Points in use:
  - A list of triplets (Local Midpoint, Remote timestamp and Roundtrip time) given in microseconds, which describes the synchronization points currently in use
- Error Approximation:
  - An approximation of the current synchronization error which is defined as the maximum roundtrip time in “Point in use” divided by two.

Synchronization process

Currently, the synchronization uses Cristian's Algorithm[5]. At fixed, growing intervals, the synchronization manager will ask the eye tracker for its time, and from that, drift and offset will be computed so that timestamps can be converted between the SDK clock and the eye tracker clock without requiring any network traffic. This is illustrated in Figure 8.

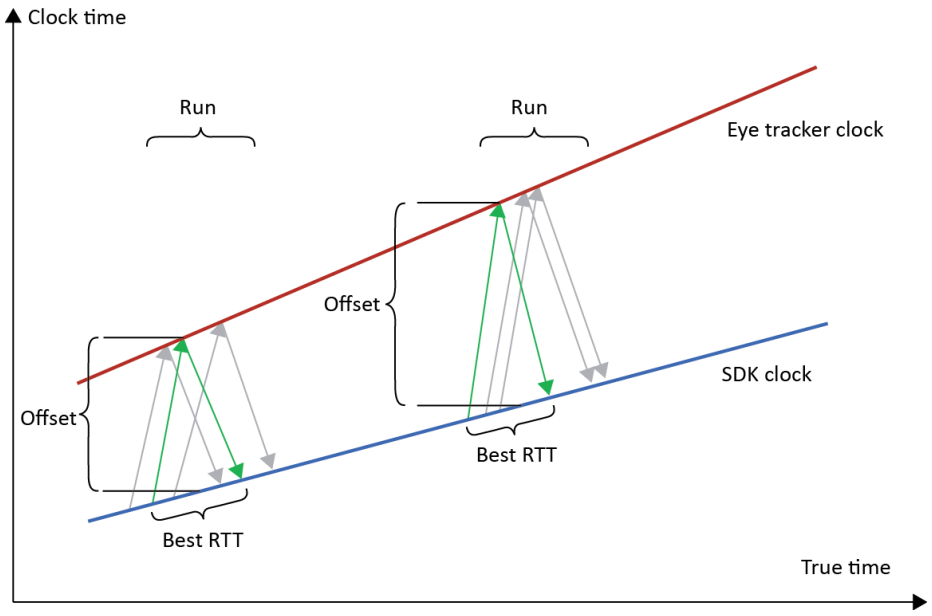


Figure 8. Synchronization procedure

At regular intervals, the synchronization manager will perform a synchronization run. From each run the synchronization point with the shortest roundtrip gets selected. A synchronization point consists of two local timestamps and one remote timestamp as shown in Figure 9.

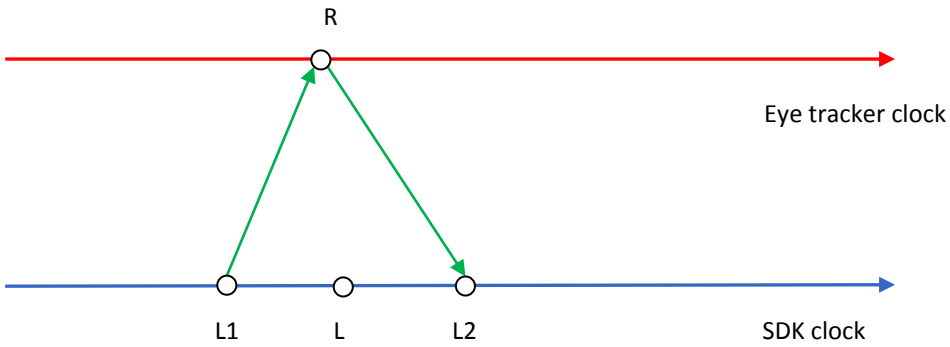


Figure 9. Synchronization Sample

From this data it is now possible to estimate the relationship between the two clocks by assuming that L=R, where L is computed as the mean value of L1 and L2. The roundtrip time (RTT=L2-L1) gives an estimate of the synchronization error in such a way that that the maximum synchronization error is bound by half the roundtrip time (RTT/2). By using two synchronization points we calculate the drift between the two clocks by using the formulas as seen below:

$$drift = \frac{r_2 - r_1}{l_2 - l_1}$$

[5] http://en.wikipedia.org/wiki/Cristian's_algorithm

r denotes the timestamp from the eye tracker clock and l is the midpoint of local start and end points (from the SDK clock). The indices indicate sync point 1 and 2.

When converting between the two clocks, the offset between them also needs to be known. By determining the offset from the first sync point you only need to take into account the drift between that point and the next when doing the conversation. Based on the assumption that L=R (as established above), the offset can be estimated as follows:

$$offset = r_1 - l_1$$

Conversion from the SDK clock to the eye tracker clock and vice versa can then be performed by:

$$t_{eyetracker} = drift \times t_{sdk} + offset$$

$$t_{sdk} = \frac{t_{eyetracker} - offset}{drift}$$

The conversion can be performed by the function convert_from_local_to_remote() and convert_from_remote_to_local() in the synchronization manager class.

Synchronization scenarios

By performing synchronization it is possible to get comparable timestamps in the gaze data and local events such as stimuli presentation and user interaction. The synchronization between one or more eye trackers and one or more client applications can be done in several different ways depending on setup. In the following pages, some commonly used scenarios will be presented. For all scenarios, one synchronization manager per client application is required for each eye tracker in the system.

One client computer and one eye tracker

The simplest case is when one client application on one client computer and one eye tracker is used as seen in Figure 10.

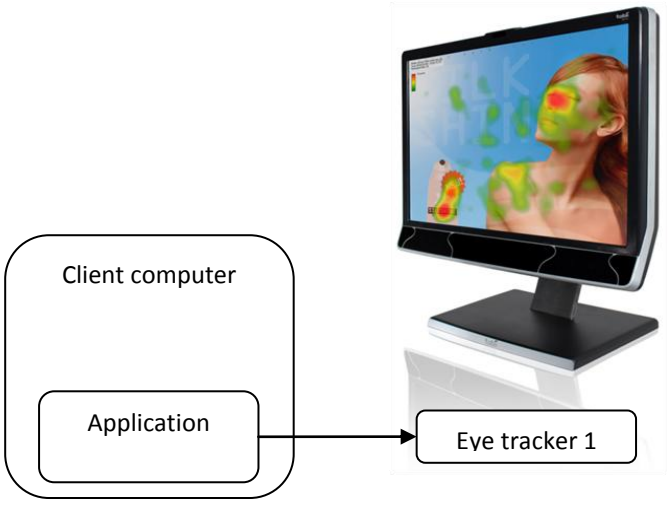


Figure 10. Standard Synchronization Scenario

In this setup there are only two clocks involved: the SDK clock in the client application and the eye tracker clock.

By using a synchronization manager it is possible to synchronize the SDK clock with the eye tracker clock. One could choose either the local SDK clock or the remote clock as the base timeline. In this case it doesn't matter which is chosen as either choice will give the same result.

One client computer and several eye trackers

A more complicated case is when there is one client application on one computer and several eye trackers as seen in Figure 11.

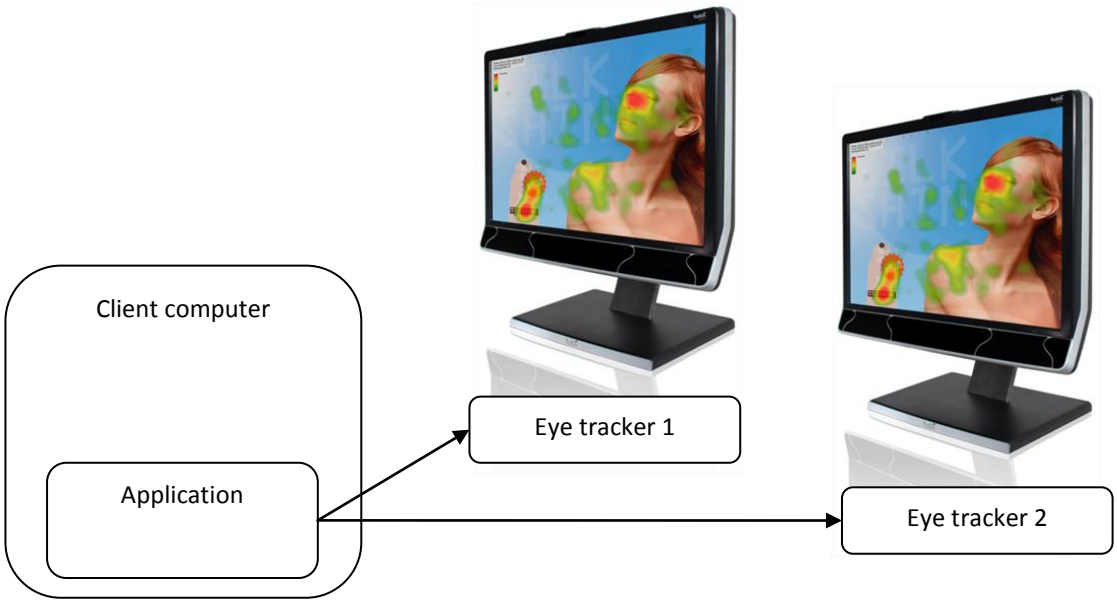


Figure 11. Synchronization Scenario with several eye trackers.

This setup increases the complexity somewhat by adding another clock that needs to be taken into account: first there is the SDK clock in the client application and then each eye tracker has its own internal clock. Therefore, in this situation there are three clocks that need to be synchronized.

To synchronize the SDK clock in the client application and the two eye trackers seen in Figure 11, two synchronization managers are needed. It makes sense to use the SDK clock as the common time source of choice since there is more than one remote clock in the system. Then it is possible to convert all timestamps from both of the eye trackers to the SDK clock. Technically it is possible to use any of the clocks as a common base. For example, one could just as well convert timestamps originating from eye tracker 1 to the time base of eye tracker 2 by first adjusting them to the local time of the client application. However, this would cause a loss of accuracy as each conversion is affected by the current synchronization error on that synchronization.

Two (or more) client computers and one eye tracker

A slightly more advanced case involves client applications on several client computers sharing the same eye tracker. This setup is shown in Figure 12.

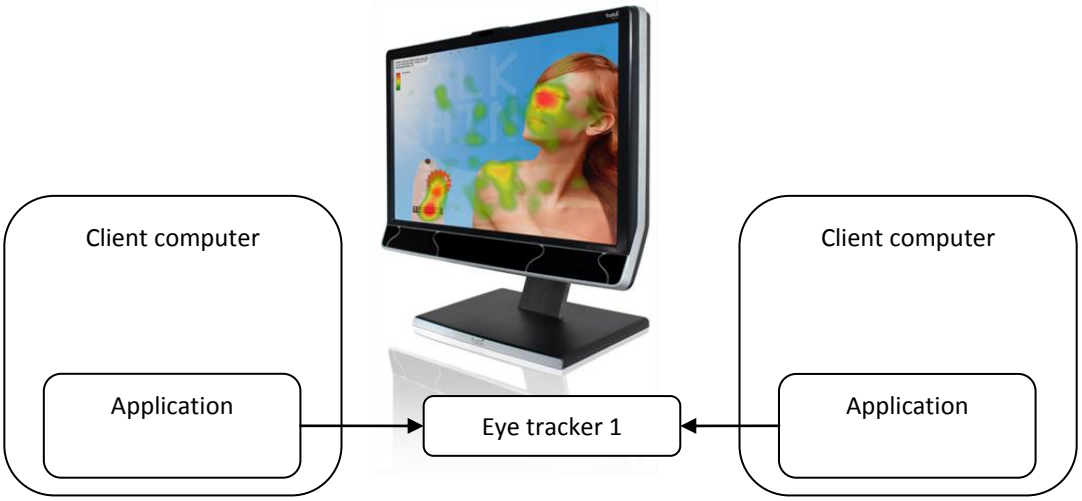


Figure 12. Synchronization Scenario with one eye tracker and several client applications.

In this setup, several clocks are also involved: there is one SDK clock per client application and one eye tracker clock.

Each client application can synchronize with the eye tracker by means of a synchronization manager. However, because the synchronization manager cannot synchronize two SDK clocks, a different approach is needed.

Each client application must use a synchronization manager to synchronize with the eye tracker's clock. In order to get comparable timestamps in all client applications they must convert their local timestamps (stimuli events) to the eye tracker's clock. This means that the eye tracker's internal clock will act as the common time base for the applications on both client computers.

Several setups of client computers and eye trackers

A really advanced case involves several client computers and one or more eye trackers as illustrated in Figure 13.

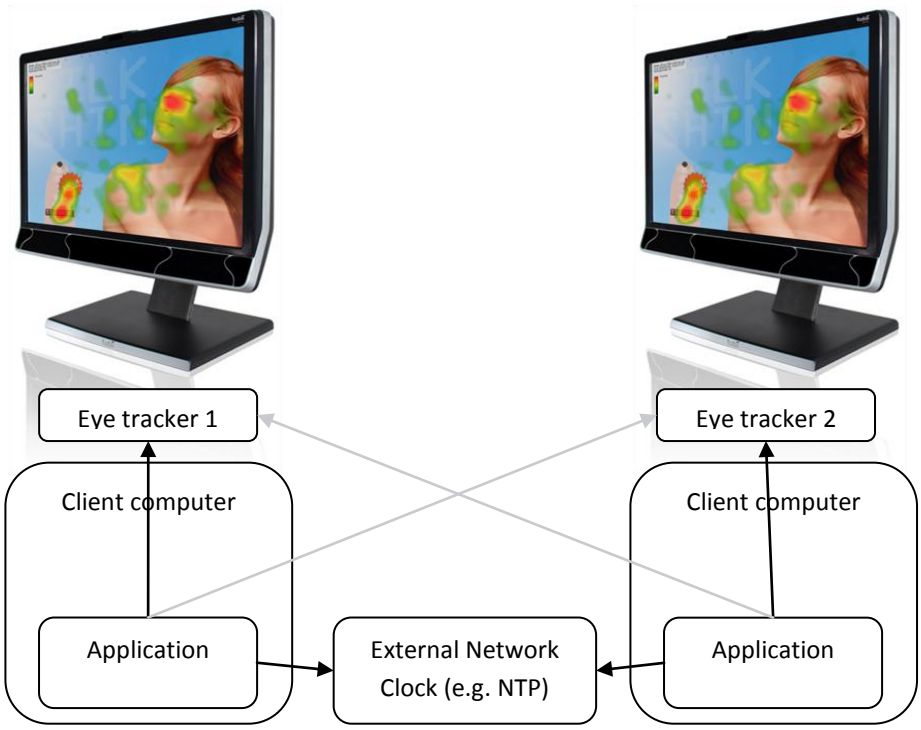


Figure 13. Synchronization Scenario with multiple eye trackers and client applications and an external clock source.

The main issue with this setup is that there is no obvious, common element that can be used as a common time base for all the other components involved.

There are two possible solutions: all applications on the client computers synchronize with all eye trackers in the system where one common eye tracker is used as the common time source (this solution uses the grey arrows in Figure 13) or a common external clock, e.g. an NTP clock, is used as the SDK clock in all client applications and as a common time source.

Getting and Setting Eye Tracker Properties

All Tobii Eye Trackers have a set of properties that can be queried, and in some cases also set. This functionality is provided by the Eyetracker class/interface and is described in the following sections.

Frame rate

All Tobii Eye Trackers supports querying for the current sampling frequency or *frame rate*. On some models, e.g., T120 and TX300, it is also possible to set the frame rate.

Track box

The track box is defined as the volume in front of the eye tracker in which tracking is possible. This information can, for example, be used to give the user feedback on her position in front of the eye tracker. Note that the track box is different between different eye tracker models and can change depending on the frame rate.

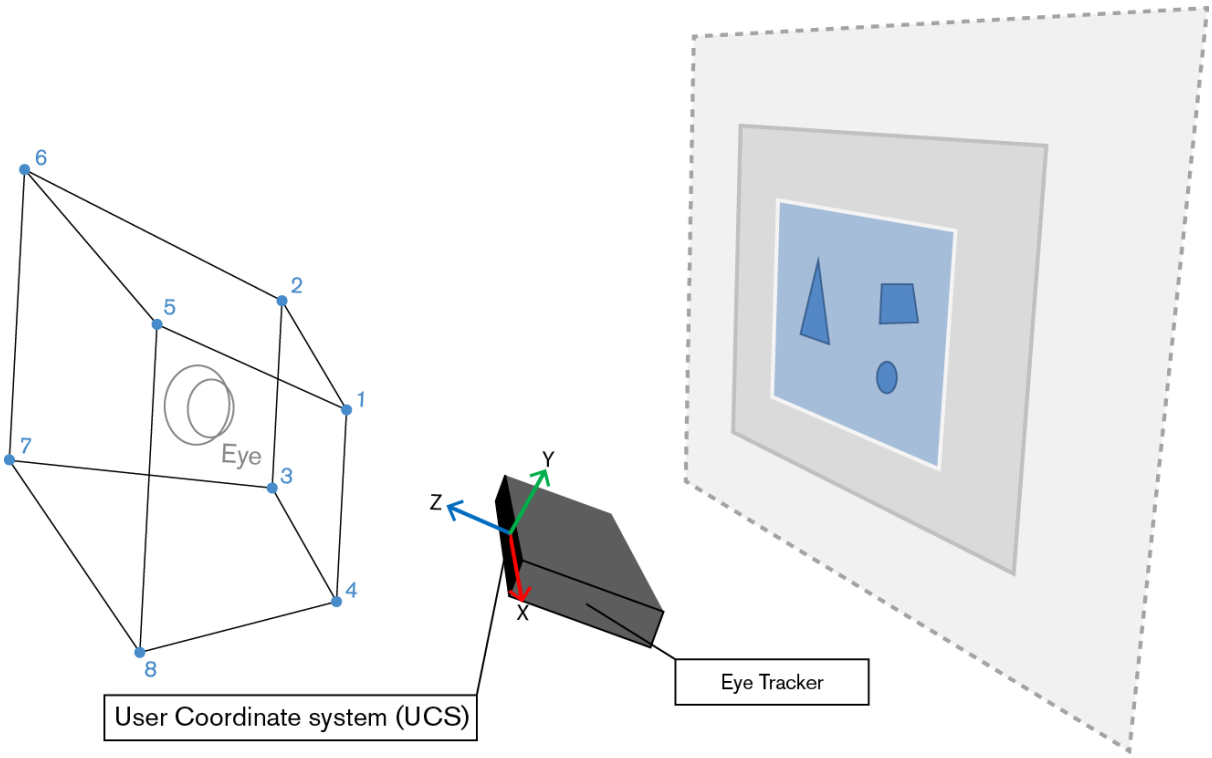


Figure 14. Schematic of the track box.

As can be seen from the schematic above (Figure 14), the track box is a frustum described by eight points. The points are reported in the User Coordinate System in millimeters (see the section *User Coordinate System (UCS)* for more information about the coordinate system).

The track box, as reported by the eye tracker, is only an approximation of the actual track box and it is only intended to be used for demonstration purposes in order to guide the user to the correct location in front of the eye tracker.

The points in the far end of the box in relation to the eye tracker (5-8) and in the near plane (1-4) have the same Z-coordinates internally. This provides the possibility to compute a relative distance between these planes taking any Z in the near and far plane and compute the relative Z coordinate as seen below:

$$Relative\ distance = \frac{Z_{eye} - Z_1}{Z_5 - Z_1}$$

Upgrading the firmware of an eye tracker

A firmware upgrade operation is quite straightforward. A Tobii firmware package file, usually with file extension .tobiipkg, is uploaded to the eye tracker and automatically installed if certain compatibility requirements are met (see Figure 15). Since the upgrade may take some time, progress is continuously reported back to the client application until the operation is completed.

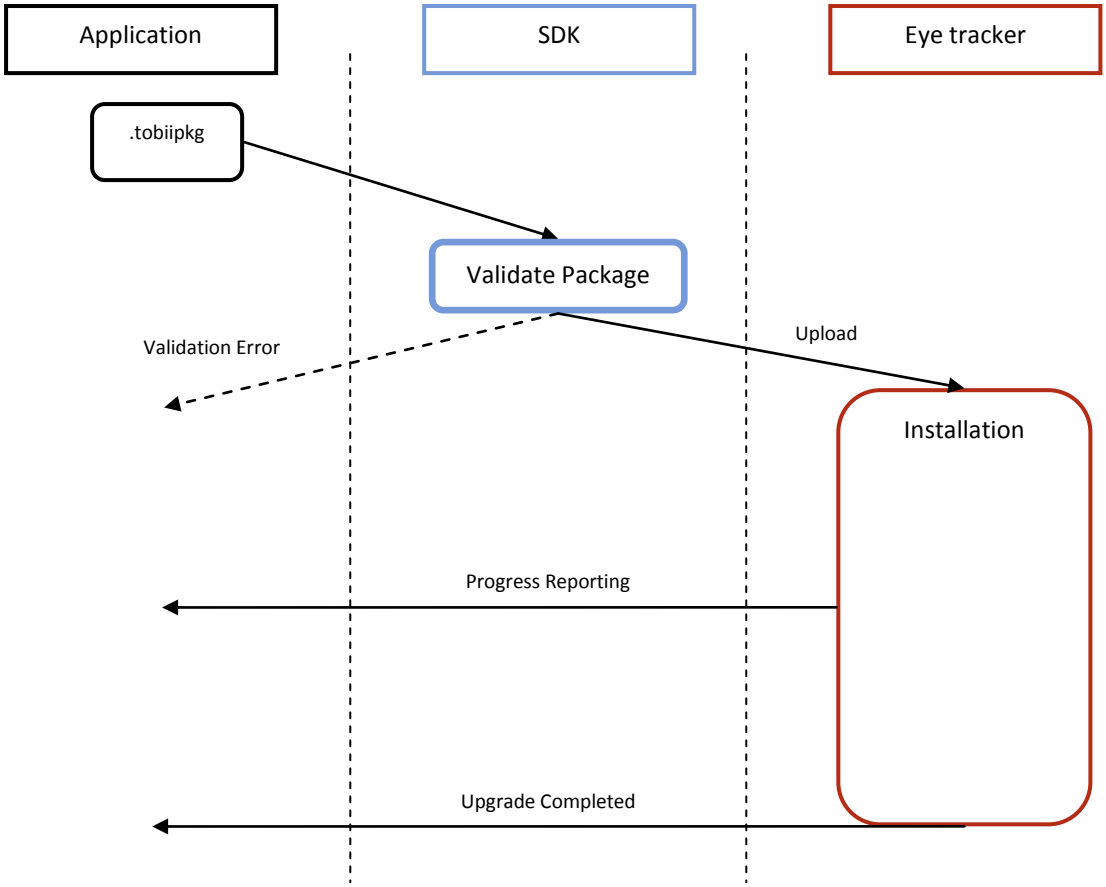


Figure 15. Upgrade Procedure

Diagnostics

The tetio component can write information about its inner workings to a log file. To enable this feature, set the following environment variables before launching the client application:

Name	Value
`TETIO_LOG_LEVEL`	Logging detail level: ERROR, WARN, INFO, or DEBUG.
`TETIO_LOG_FILE`	The log file to write to.

The .NET Framework language binding

The .NET Framework language binding consists of a single .NET assembly file called Tobii.EyeTracking.IO.dll. There is also a set of code samples written in C#, demonstrating how to use the Tobii Analytics API from .NET. The code samples are included both as source code and as executable files.

The Tobii.EyeTracking.IO assembly provides a .NET API for the functionality provided by the tetio core library, and is dependent on the core library for carrying out most of its work. Consequently, the core library must also be present in the same directory as the language binding. (It is called tetio.dll on Windows, libtetio.so on Linux, and libtetio.dylib on Mac OS X.)

The bitness of the core library must match the bitness of your application. The easiest way to make sure that this will be the case is to build the main assembly for the application, i.e., the executable file, as either an "x86" or an "x64" assembly, and to pick the corresponding version of the core library. If you build the main assembly as an "Any CPU" assembly, then it will be just-in-time compiled to the bitness of the host operating system by the .NET runtime, and it will be difficult to make sure that the correct version of the core library is selected for loading.

Using the .NET Framework language binding on Linux or Mac OS X

The SDK packages for Linux and Mac OS X do not include the .NET Framework language binding, although it is possible to also use the .NET Framework on those platforms with the help of Mono[6]. To do so, download one of the Windows SDK zip files and take the .NET Framework language binding assembly and code samples from there. The language binding assembly is built for "Any CPU", so it can be used with any bitness. After installing Mono it should be possible to make the sample files executable with "chmod +x" and then invoking them on the command line or from the File Browser or Finder.

NOTE: The *3D Eyes sample* does not work on Mono since it is built using the Windows Presentation Foundation, and Mono does not support that framework.

Code samples

Basic eyetracking sample

The *Basic eyetracking sample* is an application written using C# and Windows Forms. The example illustrates how to find eye trackers on the network, connect to a specific eye tracker, perform a calibration and subscribe to data from the eye tracker (see Figure 16).

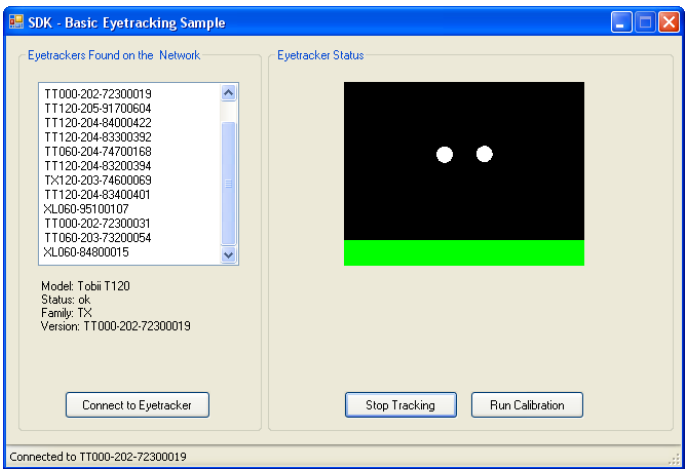


Figure 16. .NET Basic eyetracking sample

[6] See http://mono-project.com

3D Eyes sample

The *3D Eyes sample* is an application written in C# and Windows Presentation Framework (WPF). This application is a demonstration of the 3D information available in the new SDK. Having access to true eye position and gaze points in space makes it possible to create compelling visualizations (see Figure 17).

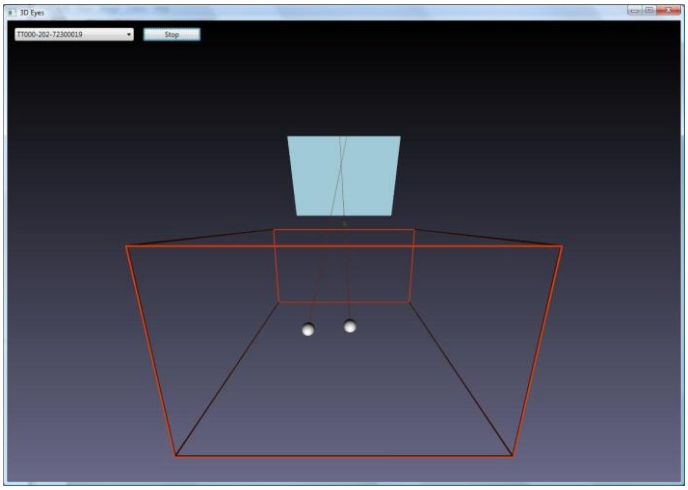


Figure 17. The .NET 3D Eyes sample showing the eyes 3D position, gaze vectors and track box

Upgrade sample

The *Upgrade sample* is an application written using C# and Windows Forms. This example illustrates the firmware upgrade functionality (see Figure 18).

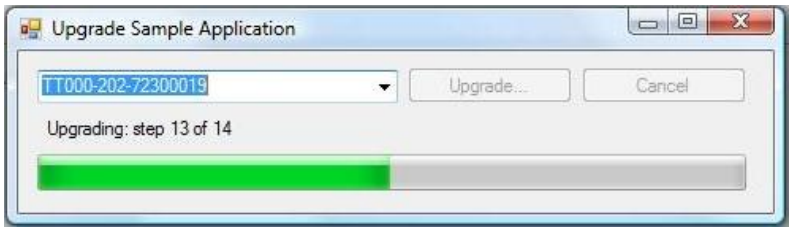


Figure 18. The .NET Upgrade Sample application

Error handling

The .NET Framework language binding uses exceptions to signal errors. The EyeTrackerException type includes an ErrorCode property that specifies what kind of error that has occurred. The possible error codes and their meanings are summarized in *Appendix 2. Error codes*.

Thread handling

Since the SDK must be able to handle data that arrives to the client in an asynchronous fashion, the core library is running multiple threads that handle reads and writes to and from the eye tracker hardware. This has some implications for applications that use the SDK.

The .NET Framework language binding uses the built-in support for multi-threading provided by the .NET Framework. All classes that expose events take a parameter of type `EventThreadingOptions` in the constructor. This is an enum type that indicates how events should be delivered to listeners and can be interpreted according to Table 3.

Table 3. EventThreadingOptions

EventThreadingOptions value	Description
`CallingThread`	All events are delivered on the thread that calls the constructor. In a single threaded application this means that all events appear on the main thread.

`BackgroundThread`	All events are delivered on a background thread. When this option is used, care must be taken to ensure that this thread doesn't manipulate the user interface, which would cause a cross thread exception to occur.

Initializing the library

The .NET Framework language binding must be initialized before use. To do this, call the Library.Init() method in your application startup code.

There is no need to un-initialize the library before exiting the application.

Class reference

Calibration class

Namespace: Tobii.EyeTracking.IO

The Calibration class is an in-memory representation of a personal calibration. It holds the raw data from the calibration as well as the calibration plot data. The raw data can be saved to persistent storage and used to re-create the calibration for download to the eye tracker at a later point in time.

Signature: public class Calibration

The Calibration class exposes the following members:

Constructors	Description
`Calibration(byte[] rawData)`	Creates a calibration object from binary data.

Properties	Description
`IList<CalibrationPlotItem> Plot { get; }`	Calibration plot data.
`byte[] RawData { get; }`	A binary representation of the calibration.

CalibrationPlotItem class

Namespace: Tobii.EyeTracking.IO

The CalibrationPlotItem class represents one gaze data sample used in a personal calibration. It is typically used for visualizing the goodness-of-fit for a calibration. This means plotting the offset between the true position (i.e. the exact position of the calibration stimulus) and the corresponding left and right eye gaze points as estimated by the eye tracker. A status code is used to indicate which of the collected samples that were incorporated in the goodness-of-fit calculations.

The status code can take the following values:

- 1: the data point is valid and has been used for the calibration.
- 0: eye tracking was successful, but the data point was not used for calibration.
- -1: eye tracking failed/invalid data point.

Signature: public class CalibrationPlotItem

The CalibrationPlotItem class exposes the following members:

Properties	Description
`Point2D TruePosition { get; set; }`	The point on the calibration plane where the calibration stimulus was displayed. The value is given in the Active Display Coordinate System.
`Point2D LeftMapPosition { get; set; }`	The left eye gaze point on the calibration plane, after calibration. The value is given in the Active Display Coordinate System.
`int LeftStatus { get; set; }`	Status code for the left eye.
`Point2D RightMapPosition { get; set; }`	The right eye gaze point on the calibration plane, after calibration. The value is given in the Active Display Coordinate System.
`int RightStatus { get; set; }`	Status code for the right eye.

Clock class

Namespace: Tobii.EyeTracking.IO

The Clock class represents a high-resolution time source, which can be used for creating timestamps and/or for synchronizing clocks with an eye tracker.

Signature: public class Clock : IClock

The Clock class exposes the following members:

Constructors	Description
`public Clock()`	Creates a default clock for the current system.
`public Clock(IClock externalClock)`	Creates a clock wrapper for the specified IClock instance.

Properties	Description
`public long Time { get; }`	The current time according to this clock. The time is given in microseconds elapsed from a constant but unspecified start time.
`public long Resolution { get; }`	The resolution of this clock in microseconds.

EyeTrackerBrowser class

Namespace: Tobii.EyeTracking.IO

The EyeTrackerBrowser class is used to browse for available eye trackers on the network and/or connected through USB.

Signature: class EyeTrackerBrowser : IEyeTrackerBrowser

The EyetrackerBrowser class exposes the following members:

Constructors	Description
`EyeTrackerBrowser()`	Creates an EyeTrackerBrowser instance.
`EyeTrackerBrowser(EventThreadingOptions opt)`	Creates an EyeTrackerBrowser instance using the specified threading option for events.

Properties	Description
`bool IsStarted { get; }`	Indicates whether browsing has been started.

Methods	Description
`void StartBrowsing()`	Starts the browser. Subscribers will not get any event notifications until the browser is started.
`void StopBrowsing()`	Stops the browser.

Events	Description
`event EventHandler<EyeTrackerInfoEventArgs> EyeTrackerFound;`	Event raised when an eye tracker appears on the network.
`event EventHandler<EyeTrackerInfoEventArgs> EyeTrackerUpdated;`	Event raised when an existing eye tracker has one or several of its properties updated.
`event EventHandler<EyeTrackerInfoEventArgs> EyeTrackerRemoved;`	Event raised when an eye tracker disappears from the network.

EyeTrackerFactory class

Namespace: Tobii.EyeTracking.IO

The EyeTrackerFactory class is used for connecting to eye trackers.

Signature: public static class EyeTrackerFactory

The EyeTrackerFactory class exposes the following members:

Methods	Description
`static IEyeTrackerFactoryInfo CreateFactoryInfoByIpAddress(string ipAddress, int serverPort, int synchonizationPort)`	Creates an IEyeTrackerFactoryInfo instance that can be used to connect to the eye tracker specified by IP address and ports. **NOTE:** Tobii Eye Trackers use server port 4455 and synchronization port 4457.

EyeTrackerInfo class

Namespace: Tobii.EyeTracking.IO

The EyeTrackerInfo class describes an eye tracker found by browsing.

Signature: public class EyeTrackerInfo : IEquatable<EyeTrackerInfo>

The EyeTrackerInfo class exposes the following members:

Properties	Description
`IEyeTrackerFactoryInfo Factory { get; }`	Gets an IEyeTrackerFactoryInfo instance for connecting to the eye tracker.
`string Generation { get; }`	Product generation of the eye tracker, e.g., “TX”.
`string GivenName { get; }`	User-defined name of the eye tracker.
`string Model { get; }`	Product model of the eye tracker, e.g., “Tobii T60”.
`string ProductId { get; }`	Product ID. This property is guaranteed to be unique among all Tobii products and can be used to identify a specific unit.
`string Status { get; }`	Current status. This will be any of the values “ok”, ”not-working”, “upgrading” or “error”.
`string Version { get; }`	Firmware version of the eye tracker.

IEyeTracker interface

Namespace: Tobii.EyeTracking.IO

The IEyeTracker interface represents an eye tracker.

Signature: public interface IEyeTracker: IDisposable

The IEyeTracker interface exposes the following members:

Methods	Description
`void AddCalibrationPoint(Point2D pt)`	Adds data to the temporary calibration buffer for one calibration point. The argument used is the point the calibrating user is assumed to be looking at and is given in the Active Display Coordinate System.
`void AddCalibrationPointAsync(Point2D pt, EventHandler<AsyncCompletedEventArgs<Empty>> responseHandler)`	Asynchronous version of the previous method. This can be useful for getting a responsive UI, since the call may take some time.
`void ClearCalibration()`	Clears the temporary calibration buffer.
`void ComputeCalibration()`	Uses the data in the temporary buffer and tries to compute calibration parameters. If the call is successful, the data is copied from the temporary buffer to the active buffer. If there is insufficient data to compute a new calibration or if the collected data is not good enough then an exception will be thrown.

`void ComputeCalibrationAsync( EventHandler<AsyncCompletedEventArgs<Empty>> responseHandler)`	Asynchronous version of the previous method. This is useful since this operation can take a while on some eye trackers.
`IList<float> EnumerateFrameRates()`	Available frame rates for this eye tracker.
`IList<string> EnumerateIlluminationModes()`	Available illumination modes for this eye tracker. This method is only applicable for TX300 and T60XL eye trackers.
`Calibration GetCalibration()`	Gets the calibration data, which is stored in the active calibration buffer. This data can be saved to a file for later use.
`float GetFrameRate()`	Gets the current frame rate. A value of, e.g., 60.0 indicates that the current frame rate is 60 Hz.
`TrackBox GetTrackBox()`	Gets the current track box. The TrackBox class is just a simple class that contains eight 3D points as described in section *Track box*.
`string GetIlluminationMode()`	Gets the current illumination mode. This method is only applicable for TX300 and T60XL eye trackers.
`bool GetLowBlinkMode()`	Gets the current low-blink mode setting. This method is only applicable for T60/T120 and X60/X120 eye trackers.
`UnitInfo GetUnitInfo()`	Gets a UnitInfo instance containing eye tracker information through the eyetracker interface
`string GetUnitName()`	Gets the eye tracker's name.
`XConfiguration GetXConfiguration()`	Gets the X configuration parameters.
`void RemoveCalibrationPoint(Point2D pt)`	Removes the data associated with a specific calibration point from the temporary calibration buffer.
`void SetCalibration(Calibration calibration)`	Sets the provided calibration, which means copying the data from the calibration into the active calibration buffer.
`void SetFrameRate(float frameRate)`	Sets the current frame rate. The frame rate value must be one of the values returned from EnumerateFramerates or this method will fail.
`void SetIlluminationMode(string illuminationMode)`	Sets the illumination mode. This method is only applicable for TX300 and T60XL eye trackers.

`void SetLowBlinkMode(bool enabled)`	Sets the low-blink mode setting. This method is only applicable for T60/T120 and X60/X120 eye trackers.
`void SetUnitName(string name)`	Sets the eye tracker's name.
`void SetXConfiguration(XConfiguration configuration)`	Sets the X configuration parameters. This method will only work on units not permanently attached to a monitor, e.g., the X120.
`void StartCalibration()`	Acquires the calibration state and clears the temporary calibration buffer.
`void StartTracking()`	Start subscribing to gaze data
`void StopCalibration()`	Releases the calibration state. This should always be done when the calibration is completed.
`void StopTracking()`	Stops gaze data subscription

Special Methods[7]	Description
`void DumpImages(Int32 count, Int32 frequency)`	Method used to diagnose hardware problems.
`void EnableExtension(Int32 extensionId)`	Enables a specific extension.
`AuthorizeChallenge GetAuthorizeChallenge(Int32 realmId, IList<Int32> algorithms)`	Gets an authorization challenge to unlock custom functionality.
`IList<Extension> GetAvailableExtensions()`	Gets a list of all available extensions.
`byte[] GetDiagnosticReport(Int32 includeImages)`	Method used to diagnose hardware problems.
`IList<Extension> GetEnabledExtensions()`	Gets a list of all enabled extensions.
`PayPerUseInfo GetPayPerUseInfo()`	Gets information whether this is a pay per use eye tracker or not.
`void ValidateChallengeResponse(Int32 realmId, Int32 algorithm, byte[] responseData)`	Validates and unlocks custom functionality.

Properties	Description
`bool RealTimeGazeData { get; set; }`	Flag affecting how gaze data is delivered to the client application. Setting the flag will ensure minimum latency at the cost of potentially dropping data packets.

[7] These methods are not recommended for use in normal application development. They are only included for the sake of completeness.

Events	Description
`event EventHandler<AsyncCompletedEventArgs> AddCalibrationPointCompleted`	Event raised when an AddCalibrationPointAsync operation is completed.
`event EventHandler<CalibrationStartedEventArgs> CalibrationStarted`	Event raised when another client acquires the calibration state.
`event EventHandler<CalibrationStoppedEventArgs> CalibrationStopped`	Event raised when another client releases the calibration state.
`event EventHandler<AsyncCompletedEventArgs> ComputeCalibrationCompleted`	Event raised when a ComputeCalibrationAsync operation is completed.
`event EventHandler<ConnectionErrorEventArgs> ConnectionError`	Event raised when the connection between the client and the eye tracker is lost. When this happens the IEyeTracker instance should be considered broken and should be disposed of.
`event EventHandler<FrameRateChangedEventArgs> FrameRateChanged`	Event raised when *another* client changes the frame rate.
`event EventHandler<GazeDataEventArgs> GazeDataReceived`	Event raised when a gaze data sample is delivered to the client application.
`event EventHandler<TrackBoxChangedEventArgs> TrackBoxChanged`	Event raised when the track box changes. This happens on some systems, e. g., on the T120, when you change the frame rate.
`event EventHandler<XConfigurationChangedEventArgs> XConfigurationChanged`	Event raised when the X configuration changes.

IClock interface

Namespace: Tobii.EyeTracking.IO

The IClock interface represents a high-resolution time source, which can be used for creating timestamps and/or for synchronizing clocks with an eye tracker.

Signature: public interface IClock

The IClock interface exposes the following members:

Properties	Description
`public long Time { get; }`	The current time according to this clock. The time is given in microseconds elapsed from a constant, but unspecified, start time.
`public long Resolution { get; }`	The resolution of this clock in microseconds.

IEyeTrackerFactoryInfo interface

Namespace: Tobii.EyeTracking.IO

The IEyeTrackerFactoryInfo interface provides methods for connecting to an eye tracker.

Signature: public interface IEyeTrackerFactoryInfo

The IEyeTrackerFactoryInfo interface exposes the following members:

Methods	Description
`IEyeTracker CreateEyeTracker()`	Connects to the eye tracker using the default threading option for events, which is CallingThread.
`IEyeTracker CreateEyeTracker(EventThreadingOptions threadingOptions)`	Connects to the eye tracker using the specified threading option for events.
`ISyncManager CreateSyncManager(Clock clock)`	Connects to the eye tracker for synchronizing clocks, using the default threading option for events, which is CallingThread.
`ISyncManager CreateSyncManager(Clock clock, EventThreadingOptions threadingOptions)`	Connects to the eye tracker for synchronizing clocks, using the specified threading option for events.

IGazeDataItem interface

Namespace: Tobii.EyeTracking.IO

The IGazeDataItem interface represents a gaze data packet received from an eye tracker.

Signature: public interface IGazeDataItem

The IGazeDataItem interface exposes the following members:

Properties	Description
`Point3D LeftEyePosition3D { get; }`	The position of the left eye in 3D space given in the User Coordinate System.
`Point3D LeftEyePosition3DRelative { get; }`	The position of the left eye in 3D space relative to the track box given in the Track Box Coordinate System.
`Point2D LeftGazePoint2D { get; }`	The gaze point on the calibration plane for the left eye given in the Active Display Coordinate System.
`Point2D LeftGazePoint3D { get; }`	The gaze point on the calibration plane for the left eye given in the User Coordinate System.
`float LeftPupilDiameter { get; }`	Diameter of the left pupil in millimeters.

`int LeftValidity { get; }`	Validity code for the left eye.
`Point3D RightEyePosition3D { get; }`	The position of the right eye in 3D space given in the User Coordinate System.
`Point3D RightEyePosition3DRelative { get; }`	The position of the right eye in 3D space relative to the track box given in the Track Box Coordinate System.
`Point2D RightGazePoint2D { get; }`	The gaze point on the calibration plane for the right eye given in the Active Display Coordinate System.
`Point2D RightGazePoint3D { get; }`	The gaze point on the calibration plane for the right eye given in the User Coordinate System.
`float RightPupilDiameter { get; }`	Diameter of the right pupil in millimeters.
`int RightValidity { get; }`	Validity code for the right eye.
`long Timestamp { get; }`	Timestamp for the point in time when the gaze data packet was sampled by the eye tracker and is given in the eye tracker clock time base. The value should be interpreted as a time interval in microseconds measured from an arbitrary point in time.

Methods	Description
`bool TryGetExtensionValue(IntegerExtensionValue id, out int value)`	Gets the value specified by the id parameter. This method is used to access extension data fields which are not provided by all eye trackers, or only provided on request. The return value is true if the requested value was successfully retrieved. The values of the IntegerExtensionValue enumeration are: *TrigSignal:* Signal sampled on the trig port of the Tobii TX-300 eye tracker.

IProgressReporter interface

Namespace: Tobii.EyeTracking.IO

The IProgressReporter interface represents an ongoing firmware upgrade operation. It can be used to display a progress indication and/or to cancel the operation.

Signature: public interface IProgressReporter

The IProgressReporter interface exposes the following members:

Methods	Description
`bool Cancel()`	Cancels the current operation.

Properties	Description
`bool CanCancel { get; }`	Indicates whether the operation can be cancelled.
`UpgradeProgress Progress { get; }`	The method provides the current estimate of the update process in terms of number of steps completed, total number of steps and % of completion of the current step in progress.

Events	Description
`event EventHandler<CanCancelEventArgs> CanCancelChanged;`	Event raised when the CanCancel property has changed.
`event EventHandler<UpgradeCompletedEventArgs> UpgradeCompleted;`	Event raised when the upgrade operation has completed.
`event EventHandler<ProgressEventArgs> UpgradeProgress;`	Event raised when the estimate of progress has changed.

ISyncManager interface

Namespace: Tobii.EyeTracking.IO

The ISyncManager interface represents an ongoing time synchronization operation between the client computer and an eye tracker.

NOTE: The IEyeTrackerFactoryInfo interface provides methods for creating ISyncManager instances.

Signature: public interface ISyncManager: IDisposable

The ISyncManager interface exposes the following members:

Methods	Description
`long LocalToRemote(long localTime)`	Converts a timestamp from local time to remote time.
`long RemoteToLocal(long remoteTime)`	Converts a timestamp from remote time to local time.

Properties	Description
`SyncState CurrentSyncState { get; }`	Describes the current state of the time synchronization operation.

Events	Description
`event EventHandler Error;`	Event raised if an error occurs during synchronization.
`event EventHandler<SyncStateChangedEventArgs> SyncStateChanged;`	Event raised when the sync state has changed.

Library class

Namespace: Tobii.EyeTracking.IO

The Library class is a static class which represents the Tobii.EyeTracking.IO library itself. It is mainly used for initialization.

Signature: public static class Library

The Library class exposes the following members:

Methods	Description
`static void Init()`	Initializes the Tobii.EyeTracking.IO library.

Properties	Description
`static bool IsInitialized { get; }`	Indicates whether the library has been initialized or not.

UpgradeManager class

Namespace: Tobii.EyeTracking.IO

The UpgradeManager class is used for upgrading the firmware of eye trackers.

Signature: public static class UpgradeManager

The UpgradeManager class exposes the following members:

Methods	Description
`static IProgressReporter BeginUpgrade(string filePath, EyeTrackerInfo unitToUpgrade, EventThreadingOptions threadingOptions)`	Starts firmware upgrade of a specified eye tracker.
`static bool UpgradePackageIsCompatibleWithDevice(string filePath, EyeTrackerInfo unitToUpgrade, out int errorCode)`	Checks whether a firmware package is compatible with the specified eye tracker.

The C++ language binding

The C++ language binding consists of a number of C++ header files. There is also a set of code samples written in C++, demonstrating how to use the Tobii Analytics API from C++. The code samples are included both as source code and as executable files.

The C++ header files provide a C++ API for the functionality provided by the tetio core library and they are dependent on the core library for carrying out most of their work. The C++ header files are stored in the "cpp/include/tobii/sdk/cpp" folder.

In Windows, the core library consists of an import library, tetio.lib, and a runtime library tetio.dll. For Linux and Mac OS X, the core library is a shared object library, libtetio.so (Linux) or libtetio.dylib (Mac OS X). The core library is stored in the "cpp/lib" folder.

When building for Windows or Linux, the bitness, 32- or 64-bit, of the core library must match the bitness of your application. For Mac, the core library is a fat (universal) binary and may be used to build for any bitness.

The C++ language binding is dependent on boost, which must be installed to be able to build. The C++ language binding is designed with boost 1.49, but should be compatible with earlier as well as later boost versions.

Code samples

Tobiictl sample

The *tobiictl* is a sample console application (see Figure 19). It is a command line tool that lets the user find eye trackers on the network. For a specific eye tracker it is possible to get information, connect, perform a calibration and subscribe to data.

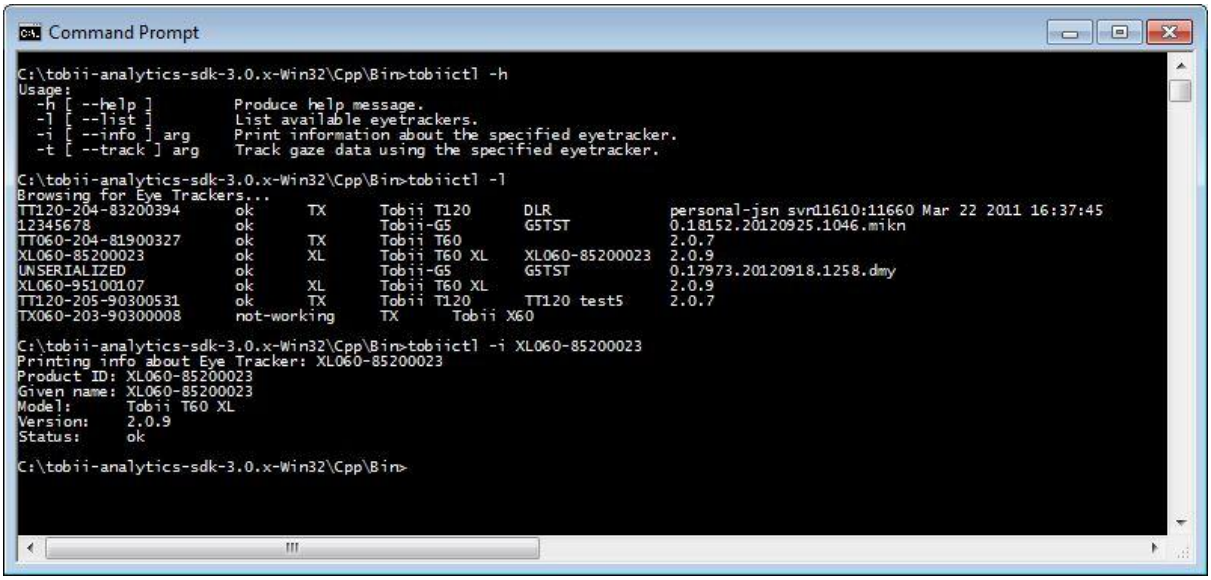


Figure 19. tobiictl, a C++ sample console application

QtEyeTrackingSample

The *QtEyeTrackingSample* is an application written using Qt. The example illustrates how to find eye trackers on the network, connect to a specific eye tracker, perform a calibration and subscribe to data from the eye tracker (see Figure 20).

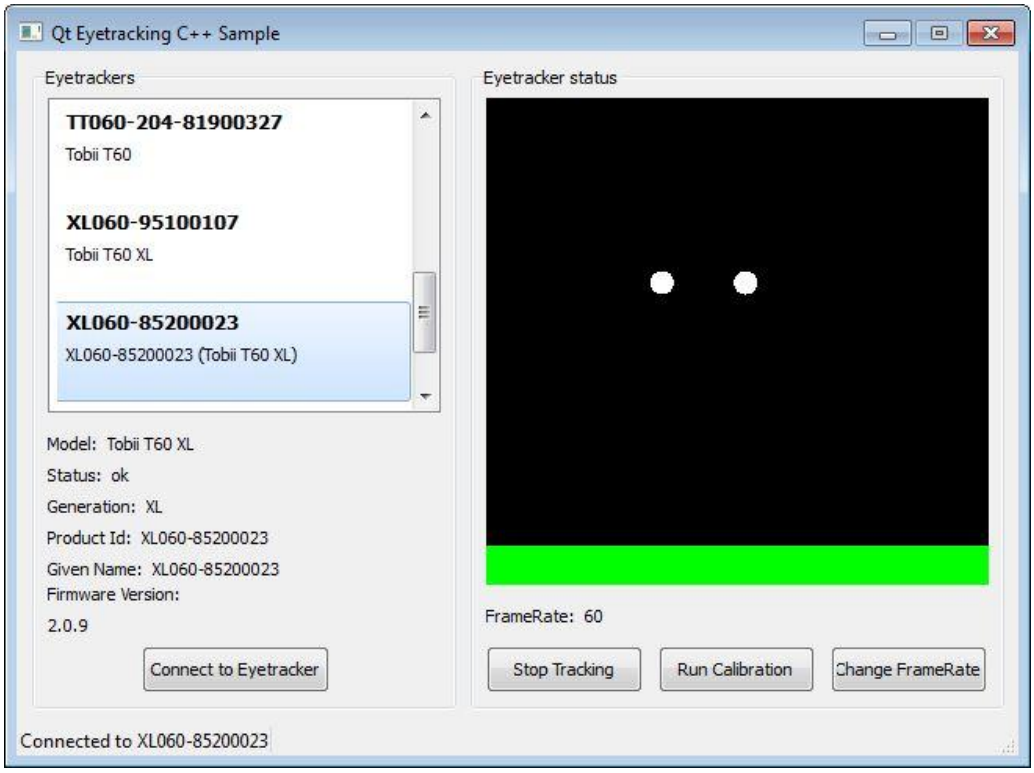


Figure 20. QtEyeTrackingSample a C++ sample Qt application

Error handling

The C++ language binding uses exceptions to signal errors. The EyeTrackerException type includes an ErrorCode that specifies the kind of error that occurred. The possible error codes and their meanings are summarized in *Appendix 2. Error codes*.

Thread handling

Since the SDK must be able to handle data that arrives to the client in an asynchronous fashion, the core library runs multiple threads that handle reads and writes to and from the eye tracker hardware. This has some implications for applications that use the SDK.

In the C++ language binding, all events are delivered on a background thread. This thread is provided by the user in the form of a MainLoop object. The MainLoop object is created by the client application and provided as an argument to all the constructors that need one. Typically, the SDK client application creates an instance of MainLoop and keeps it alive for the duration of the application. The MainLoop class is not attached to a thread when it is created. Therefore, the application has to first create a thread and then call the MainLoop::run() method. This call will block until someone calls MainLoop::quit(). When that happens the thread will return from MainLoop::run() and continue the execution. It is important not to call back into the SDK from the background thread since that may cause deadlocks. Instead, events should be marshaled to the main thread (or possibly another thread). This can usually be accomplished by using a platform specific library, e.g., MFC on Windows or gtkmm in Linux.

Initializing the library

The C++ language binding must be initialized before use. To do this, call the Library.init() method in your application startup code.

There is no need to un-initialize the library before exiting the application.

Class reference

The complete C++ language binding lies in the same name space, tobii.sdk.cpp.

Calibration class

The Calibration class is an in-memory representation of a personal calibration. It holds the raw data from the calibration as well as the calibration plot data. The raw data can be saved to persistent storage and used to re-create the calibration for download to the eye tracker at a later point in time.

Signature: class Calibration

The Calibration class exposes the following members:

Methods	Description
`CalibrationPlotItem::plot_data_vector_t getPlotData()`	Gets the calibration plot data contained in this calibration instance.
`Blob& getRawData()`	Gets a binary representation of the calibration. This data can be saved to file and used at a later point in time.

CalibrationPlotItem struct

The CalibrationPlotItem struct represents one gaze data sample used in a personal calibration. It is typically used for visualizing the goodness-of-fit for a calibration. This means plotting the offset between the true position (i.e. the exact position of the calibration stimulus) and the corresponding left and right eye gaze points as estimated by the eye tracker. A status code is used to indicate which of the collected samples that were incorporated in the goodness-of-fit calculations.

The status code can take the following values:

- 1: the data point is valid and has been used for the calibration.
- 0: eye tracking was successful, but the data point was not used for calibration.
- -1: eye tracking failed/invalid data point.

Signature: struct CalibrationPlotItem

The CalibrationPlotItem struct exposes the following members:

Variables	Description
`Point2D truePosition`	The point on the calibration plane where the calibration stimulus was displayed. The coordinates are given in the Active Display Coordinate System.
`Point2D leftMapPosition`	The point of the left eye gaze point on the calibration plane, after calibration, given in the Active Display Coordinate System.
`int32_t leftStatus`	Status code for the left eye.

`Point2D rightMapPosition`	The right eye gaze point on the calibration plane, after calibration, given in the Active Display Coordinate System.
`int32_t rightStatus`	Status code for the right eye.

Clock class

The Clock class represents a high-resolution time source, which can be used for creating timestamps and/or for synchronizing clocks with an eye tracker.

Signature: class Clock

The Clock class exposes the following members:

Constructors	Description
`Clock()`	Creates an instance of the default clock.
`Clock(const Clock& other)`	Copy constructor

Methods	Description
`int64_t getTime()`	Gets the current time as a microsecond value.
`int64_t getResolution()`	Gets the clock resolution as a microsecond value.
`tobii_sdk_clock_t* getHandle()`	

EyeTracker class

The EyeTracker class represents an eye tracker.

Signature: class EyeTracker

The EyeTracker class exposes the following members:

Methods	Description
`void setFrameRate(float framerate)`	Sets the current frame rate. The frame rate value must be one of the values returned from EnumerateFramerates or this method will fail.
`float getFrameRate()`	Gets the current frame rate. A value of, e.g., 60.0 indicates that the current frame rate is 60 Hz.
`std::vector<float> enumerateFrameRates()`	Gets a list of possible frame rates for this eye tracker.
`uint32_t getLowblinkMode()`	Gets the current low-blink mode setting. This method is only applicable for T60/T120 and X60/X120 eye trackers.

`void setLowblinkMode(uint32_t enabled)`	Sets the current low-blink mode setting. This method is only applicable for T60/T120 and X60/X120 eye trackers.
`void setUnitName(const std::string& name)`	Sets the eye tracker's name.
`std::string getUnitName()`	Gets the eye tracker's name.
`UnitInfo::pointer_t getUnitInfo()`	Gets a UnitInfo instance containing eye tracker information through the eyetracker interface.
`void startCalibration()`	Acquires the calibration state and clears the temporary calibration buffer.
`void stopCalibration()`	Releases the calibration state. This should always be done when the calibration is finished!
`void clearCalibration()`	Clears the temporary calibration buffer.
`void addCalibrationPoint(` `  const Point2d& point)`	Adds data to the temporary calibration buffer for one calibration point. The argument used is the point the calibrating user is assumed to be looking at and is given in the Active Display Coordinate System.

`void addCalibrationPointAsync(`
`  const Point2d& point,`
`  const async_callback_t& completedHandler)`
	Asynchronous version of the previous method. This can be useful to get a responsive UI, since the call may take some time.

`void removeCalibrationPoint(` `  const Point2d& point)`	Removes the data associated with a specific calibration point from the temporary calibration buffer.
`void computeCalibration()`	Uses the data in the temporary buffer and tries to compute calibration parameters. If the call is successful, data is copied from the temporary buffer to the active buffer. If there is insufficient data to compute a new calibration or if the collected data is not good enough then an exception will be thrown.
`void computeCalibrationAsync(` `  const async_callback_t& completedHandler)`	Asynchronous version of the previous method, since this operation can take a while on some eye trackers.
`Calibration::pointer_t getCalibration()`	Gets the calibration stored in the active calibration buffer. This data can be saved to file for later use.

`void setCalibration(` `  Calibration::pointer_t calibration)`	Sets the provided calibration, which means copying the data from the temporary calibration buffer into the active calibration buffer.
`void startTracking()`	Start subscribing to gaze data
`void stopTracking()`	Stops gaze data subscription
`TrackBox::pointer_t` `  getTrackBox()`	Gets the current track box. Returns a TrackBox instance containing eight 3D points given in the User Coordinate System.
`void setXConfiguration(` `  const XConfiguration& configuration)`	Sets the X configuration parameters. This method will only work on x-units, e.g., the X120.
`XConfiguration::pointer_t` `getXConfiguration()`	Gets the X configuration parameters.
`std::string getIlluminationMode()`	Gets the current illumination mode. This method is only applicable for TX300 and T60XL eye trackers.
`void setIlluminationMode(` `  const std::string& illuminationMode)`	Sets the illumination mode. This method is only applicable for TX300 and T60XL eye trackers.
`std::vector<std::string>` `enumerateIlluminationModes()`	Available illumination modes for this eye tracker. This method is only applicable for TX300 and T60XL eye trackers.

`connection_t addFrameRateChangedListener(`
`  const`
`framerate_changed_event_t::slot_type&`
`listener)`
	Adds a listener for the frame rate changed event. This event is fired when *another* client changes the frame rate.

`connection_t addCalibrationStartedListener(`
`  const empty_event_t::slot_type&`
`listener)`
	Adds a listener to the calibration started event. This event is fired when another client acquires the calibration state.

`connection_t addCalibrationStoppedListener(`
`  const empty_event_t::slot_type&`
`listener)`
	Adds a listener to the calibration stopped event. This event is fired when another client releases the calibration state.

`connection_t addGazeDataReceivedListener(`
`  const`
`gaze_data_received_event_t::slot_type&`
`listener)`
	Adds a listener for the gaze data event. This event occurs when a gaze data sample is delivered to the client application.

`connection_t addTrackboxChangedListener(`
`  const empty_event_t::slot_type&`
`listener)`
	Adds a listener for the track box changed event. This happens on some systems, e. g., on the T120, when you change the frame rate.

`connection_t addXConfigChangedListener(const x_configuration_changed_event_t::slot_type& listener)`	Adds a listener for the X configuration changed event. This event is fired when the X configuration changes.
`connection_t addConnectionErrorListener(const connection_error_event_t::slot_type& listener)`	This event is fired when network connectivity is lost. When this happens the eye tracker instance should be considered broken and should be disposed of.

Special Methods[8]	Description
`void dumpImages( uint32_t count, uint32_t frequency)`	Method used to diagnose hardware problems.
`void enableExtension(uint32_t extensionId)`	Enables a specific extension.
`AuthorizeChallenge getAuthorizeChallenge( uint32_t realmId, const std::vector<uint32_t> algorithms)`	Gets an authorization challenge to unlock custom functionality.
`Extension::vector_pointer_t getAvailableExtensions()`	Gets a list of all enabled extensions.
`Blob getDiagnosticsReport( uint32_t includeImages)`	Method used to diagnose hardware problems.
`Extension::vector_pointer_t getEnabledExtensions()`	Gets a list of all enabled extensions.
`PayPerUseInfo getPayPerUseInfo()`	Gets information whether this is a pay per use eye tracker or not.
`void validateChallengeResponse( uint32_t realmId, uint32_t algorithm, Blob& responseData)`	Validates and unlocks custom functionality.

EyeTrackerBrowser class

The EyeTrackerBrowser class is used to browse for available eye trackers on the network and/or connected through USB.

The user registers as listener for browser events and will receive callbacks when an eye tracker is found, updated or removed. The callback contains an event type (TRACKER_FOUND, TRACKER_UPDATED, TRACKER_REMOVED) and a corresponding EyeTrackerInfo object.

Signature: class EyeTrackerBrowser

The EyetrackerBrowser class exposes the following members:

Methods	Description
`void addEventListener(const browser_event_t::slot_type& listener)`	Adds a browser callback event listener to this Eye tracker browser instance.

[8] These methods are not recommended for use in normal application development. They are only included for the sake of completeness.

`void start()`	Starts the browser. Subscribers will not get any event notifications until the browser is started.
`void stop()`	Stops the browser.

EyeTrackerBrowserFactory class

The EyeTrackerBrowserFactory class is used for creating the EyeTrackerBrowser and FactoryInfo class.

Signature: class EyeTrackerBrowserFactory

The EyeTrackerBrowserFactory class exposes the following members:

Methods	Description
`static EyeTrackerBrowser::pointer_t createBrowser(MainLoop& mainLoop)`	Returns an EyeTrackerBrowser instance that can be used to browse for eye trackers.
`static FactoryInfo::pointer_t createFactoryInfoByIpAddress( const std::string& ipAddress, uint32_t tetserverPort, uint32_t syncPort)`	Returns a FactoryInfo instance that can be used to connect to the eye tracker specified by IP address and ports. **NOTE:** Tobii Eye Trackers use server port 4455 and synchronization port 4457.
`static FactoryInfo::pointer_t createFactoryInfoByName( const std::string& trackerName, uint32_t tetserverPort, uint32_t syncPort)`	Returns a FactoryInfo instance that can be used to connect to the eye tracker specified by host name and ports. **NOTE:** Tobii Eye Trackers use server port 4455 and synchronization port 4457.

EyeTrackerInfo class

The EyeTrackerInfo class describes an eye tracker found by browsing.

Signature: class EyeTrackerInfo

The EyeTrackerInfo class exposes the following members:

Methods	Description
`std::string getProductId()`	Gets the product ID property as a string
`std::string getGivenName()`	Gets the given name property as a string
`std::string getModel()`	Gets the model property as a string
`std::string getGeneration()`	Gets the generation property as a string
`std::string getVersion()`	Gets the version property as a string
`std::string getStatus()`	Gets the status property as a string. This will be any of the values “ok”, ”not-working”, “upgrading” or “error”.
`FactoryInfo::pointer_t getFactoryInfo()`	Gets a FactoryInfo object from this EyetrackerInfo. This object is used when connecting to an eye tracker.

`ProgressReporter::pointer_t beginUpgrade (UpgradePackage& upgradePackage, MainLoop& mainLoop)`	Starts the upgrade. The process is asynchronous, which means that this call will return immediately. Returns a ProgressReporter instance that will report when the upgrade procedure fails or completes.
`bool isCompatible( UpgradePackage& upgradePackage, MainLoop& mainLoop, uint32_t& errorCode)`	Checks if an upgrade package is compatible with a specific eye tracker.

EyeTrackerFactory class

The EyeTrackerFactory class provides methods for connecting to an eye tracker.

Signature: class EyeTrackerFactory

The EyeTrackerFactory class exposes the following members:

Methods	Description
`std::string getRepresentation()`	Returns a textual representation of the eye tracker address.
`EyeTracker::pointer_t createEyeTracker( MainLoop& mainLoop)`	Connects to the eye tracker and returns an EyeTracker instance.
`SyncManager::pointer_t createSyncManager(Clock& clock, MainLoop& mainLoop)`	Connects to the eye tracker for synchronizing clocks and returns a SyncManager instance.

GazeDataItem class

The GazeDataItem interface represents a gaze data packet received from an eye tracker.

Signature: class GazeDataItem

The GazeDataItem class exposes the following members:

Variables	Description
`uint64_t timestamp`	Timestamp for the point in time when the gaze data packet was sampled by the eye tracker and is given in the eye tracker clock time base. The value should be interpreted as a time interval in microseconds measured from an arbitrary point in time
`Point3D leftEyePosition3d` `Point3D rightEyePosition3d`	The position of the left/right eye in 3D space. User Coordinate system.
`Point3D leftEyePosition3dRelative` `Point3D rightEyePosition3dRelative`	The position of the left/right eye in 3D space relative to the track box given in the Track Box Coordinate System.

`Point2D leftGazePoint2d` `Point2D rightGazePoint2d`	The gaze point on the calibration plane for the left/right eye. Active Display Coordinate System.
`Point2D leftGazePoint3d` `Point2D rightGazePoint3d`	The gaze point on the calibration plane for the left/right eye. User Coordinate System.
`float leftPupilDiameter` `float rightPupilDiameter`	Diameter of the left/right pupil in millimeters.
`int leftValidityint` `rightValidity`	Validity code for the left/right eye.

Methods	Description

`bool tryGetExtensionValue(`
`  extension_value_uint32_t extension,`
`  uint32_t& value)`
	Gets the value specified by the id parameter. This method is used to access extension data fields that are not provided by all eye trackers, or only provided on request. The return value is true if the requested value was successfully retrieved.
	
	The values of the GazeDataItem::`extension_value_uint32_t` enumeration are:
	
	`EXTENSION_TRIG_SIGNAL:` Signal sampled on the trig port of the Tobii TX-300 eye tracker.

Library class

The Library class is a static class that represents the tetio library itself. It is mainly used for initialization.

Signature: class Library

The Library class exposes the following members:

Methods	Description
`static void init()`	Initializes the tetio library.

MainLoop class

The MainLoop class is responsible for keeping the main loop thread alive.

Signature: class MainLoop

The MainLoop class exposes the following members:

Constructors	Description
`MainLoop()`	Creates a new MainLoop instance.
`MainLoop(const MainLoop& other)`	Copy constructor

Methods	Description
`void run()`	Starts the main loop thread. This call is blocking and should be called on a background thread. All asynchronous events will be delivered on this thread.
`void quit()`	Tell the main loop thread to stop doing work and return execution from the blocking call to run(). This is typically done when the application exits.
`tobii_sdk_mainloop_t* getHandle()`	Return a handle to the main loop.

ProgressReporter class

The ProgressReporter class represents an ongoing firmware upgrade operation. It can be used to display a progress indication and/or to cancel the operation

Signature: class ProgressReporter

The ProgressReporter class exposes the following members:

Methods	Description
`bool canCancel()`	Indicates whether the operation can be cancelled.
`bool cancel()`	Cancels the current operation.
`Progress getProgress()`	Current estimate of progress.
`connection_t addUpgradeCompletedListener( const upgrade_completed_event_t::slot_type& listener)`	Event raised when the upgrade operation has completed.
`connection_t addProgressChangedListener( const upgrade_progress_event_t::slot_type& listener)`	Event raised when the estimate of progress has changed.
`connection_t addCanCancelChangedListener( const can_cancel_changed_event_t::slot_type& listener)`	Event raised when the CanCancel property has changed.

SyncManager class

The SyncManager class represents an ongoing time synchronization operation between the host computer and an eye tracker.

NOTE: The EyeTrackerFactory class provides methods for creating SyncManager instances.

Signature: class SyncManager

The SyncManager class exposes the following members:

Methods	Description
`int64_t localToRemote(int64_t localTime)`	Converts from local to remote (eye tracker) time. The time unit is always microseconds.
`int64_t remoteToLocal(int64_t remoteTime)`	Converts from remote (eye tracker) time to local time. The time unit is always microseconds.
`SyncState::pointer_t getSyncState()`	Gets the current synchronization state in the form of a SyncState instance.
`connection_t addSyncStateChangedListener( const sync_state_changed_event_t::slot_type& listener)`	Adds a listener to the sync state changed event, which is fired when the synchronization state is updated. The event argument contains the updated sync state information similar to what you get from the get_sync_state() method.

SyncState class

The SyncState class holds the current state of the time synchronization operation.

Signature: class SyncState

The SyncState class exposes the following members:

Methods	Description
`sync_state_flag_t getSyncStateFlag()`	Describes the current state of the time synchronization operation which may be `UNSYNCHRONIZED, STABILIZING or SYNCHRONIZED`
`const std::vector<SyncPoint>& getSyncPoints()`	Returns a list of tuples with three values in them, i.e. *local midpoint, remote timestamp*, *remote point timestamp* and *roundtrip time*.

UpgradePackage class

The UpgradePackage represents an eye tracker firmware upgrade package.

NOTE: The Upgrade process is started by using the EyeTrackerBrowserFactory class.

Signature: class UpgradePackage

The UpgradePackage class exposes the following members:

Constructors	Description
`UpgradePackage(const uint8_t* packageData, uint32_t packageSize)`	Creates a new UpgradePackage instance with size and content according to supplied arguments.
`UpgradePackage(const UpgradePackage& other)`	Copy constructor

Methods	Description
`tobii_sdk_upgrade_package_t* getHandle()`	Returns a handle to the upgrade package representation in the core library.

The Matlab language binding

The Matlab language binding consists of several functions, each located in separate m-files. The language binding also includes a mex file called tetio_matlab.mex<architecture>. All calls from the m-file function wrappers are sent to the mex file, which performs the requested actions. Users should only call and use the provided m-file functions. There is also a code sample demonstrating how to use these functions in Matlab.

The only installation necessary for the Matlab language binding is to copy the Matlab directory from the SDK top directory to a known location on the local drive. The path to the Matlab directory will hereafter be called <tetio_matlab_dir>.

The path to the directory <tetio_matlab_dir>\tetio must be provided in the .m-file before any functions in the Matlab language binding can be called. This is done by using the command addpath('<tetio_matlab_dir>\tetio'). If using the Matlab Command Window to call functions, navigate to the <tetio_matlab_dir> and give the command "addpath tetio" or provide the full path as addpath('<tetio_matlab_dir>\tetio').

NOTE: There are both 32 and 64-bit versions of the mex file available on the Windows platform. For the Mac OSX platform, only a 64-bit version is available and it works exclusively with a 64-bit version of Matlab. There is currently no version of the mex file available for Linux.

Code samples

To run a sample, start Matlab and open a Matlab Command Window. In the command window, locate and open the folder containing the sample you want to run (e.g. type "cd C:\<tetio_matlab_dir>\EyeTrackingSample"). When in the correct folder, type the name of the sample file excluding the file ending ".m" and the sample will start.

EyeTrackingSample

The *EyeTrackingSample* is an application written in Matlab. The example illustrates how to find eye trackers on the network, connect to a specific eye tracker, perform a calibration and read data from the eye tracker.

Error handling

The Matlab language binding uses mexError to signal errors. The mexError function throws an exception to the running Matlab code. The exception may contain an error code. Error codes are described in *Appendix 2. Error codes*.

Initializing the library

The Matlab language binding must be initialized before use. To do this, call the tetio_init() function in your application startup code.

There is a function available for un-initializing the library, but the use of it is optional.

Interacting with the Matlab language binding

The Matlab language binding consists of several functions located in m-files. Call these functions to interact with an eye tracker.

Function reference

Methods	Description
`tetio_init()`	Initializes the library and makes it possible to call other functions.
`tetio_cleanUp()`	Cleans up the current instantiation of the library.
`EyeTrackerInfoStruct array = tetio_getTrackers()`	Returns information about all eye trackers found on the network.
`tetio_connectTracker(productId)`	Connects to the specified eye tracker.
`tetio_disconnectTracker()`	Disconnects from the currently connected eye tracker.
`frameRateArray = tetio_enumerateFrameRates()`	Returns an array of available frame rates.
`tetio_setFrameRate(framerate)`	Sets the frame rate.
`framerate = tetio_getFrameRate()`	Returns the current frame rate.
`tetio_stopTracking()`	Stops tracking.
`tetio_startTracking()`	Starts tracking.
`gazeDataArray = tetio_readGazeData()`	Reads the available data and returns it. The returned data is in the form of arrays.
`tetio_addCalibPoint(x, y)`	Adds data to the temporary calibration buffer for one calibration point. The argument used is the point the calibrating user is assumed to be looking at and is given in the Active Display Coordinate System.
`tetio_removeCalibPoint(x, y)`	Removes a calibration point.
`tetio_clearCalib()`	Clears all calibration points.
`tetio_computeCalib()`	Computes the previous calibration run. If there is insufficient data to compute a new calibration, or if the collected data is not good enough, then an exception `will` be thrown.
`calibrationPlotArray = tetio_getCalibPlotData()`	Returns the result of the last calibration.
`tetio_startCalib()`	Starts calibration.
`tetio_stopCalib()`	Stops calibration.
`tetio_clockSyncState()`	Describes the current state of the time synchronization operation which may be 0 (if not

	synchronized) or 1 (if synchronized).
tetio_localClockResolution()	Returns the resolution of the local clock in microseconds.
tetio_localTimeNow()	Returns the time of the local clock in microseconds.
tetio_localToRemoteTime()	Converts from local to remote (eye tracker) time. The time unit is always microseconds and given.
tetio_remoteToLocalTime()	Converts from remote (eye tracker) time to local time. The time unit is always microseconds.

Time synchronization functions

NOTE: You should only call time synchronization functions after having done a successful call to `tetio_connectTracker`.

Methods	Description
`tetio_clockSyncState()`	Describes the current state of the time synchronization operation which may be 0 (if not synchronized) or 1 (if synchronized).
`tetio_localToRemoteTime(localTime)`	Converts from local to remote (eye tracker) time. The time unit is always microseconds and given in the int64 format.
`tetio_remoteToLocalTime(remoteTime)`	Converts from remote (eye tracker) time to local time. The time unit is always microseconds and given in the int64 format.
`tetio_localTimeNow()`	Returns the time of the local clock in microseconds in the int64 format.
`tetio_localClockResolution()`	Returns the resolution of the local clock in microseconds given in the int64.

Return types

EyeTrackerInfoStruct

The EyeTrackerInfoStruct describes an eye tracker on the network.

Type	Field	Description
char	**ProductId**	This property is guaranteed to be unique among all Tobii products and can be used to identify a specific unit.
char	**Model**	Product model of the eye tracker, e.g., “Tobii T60”.
char	**GivenName**	User-defined name of the eye tracker.
char	**Generation**	Product generation of the eye tracker, e.g., “TX”.

char	**FirmwareVersion**	Firmware version of the eye tracker.
char	**Status**	Current status. This will be any of the values “ok”, ”not-working”, “upgrading” or “error”.

CalibrationPlotArray

The CalibrationPlotArray represents one gaze data sample used in a personal calibration. It is typically used for visualizing the goodness-of-fit for a calibration. This means plotting the offset between the true position (i.e. the exact position of the calibration stimulus) and the corresponding left and right eye gaze points as estimated by the eye tracker. A status code is used to indicate which of the collected samples were incorporated in the goodness-of-fit calculations.

The status code can take the following values:

- 1: the data point is valid and has been used for the calibration.
- 0: eye tracking was successful, but the data point was not used for calibration.
- -1: eye tracking failed/invalid data point.

The CalibrationPlotArray contains the following values:

Type	Field	
`double`	`truePosition.x`	The X coordinate on the calibration plane where the calibration stimulus was displayed. The coordinate is given in the Active Display Coordinate System.
`double`	`truePosition.y`	The Y coordinate on the calibration plane where the calibration stimulus was displayed. The coordinate is given in the Active Display Coordinate System
`double`	`leftMapPosition.x`	The X coordinate of the left eye gaze point on the calibration plane, after calibration. The value is given in the Active Display Coordinate System.
`double`	`leftMapPosition.y`	The Y coordinate of the left eye gaze point on the calibration plane, after calibration. The value is given in the Active Display Coordinate System.
`double`	`leftStatus`	Status code for the left eye.
`double`	`rightMapPosition.x`	The X coordinate of the right eye gaze point on the calibration plane, after calibration. The value is given in the Active Display Coordinate System.
`double`	`rightMapPosition.y`	The Y coordinate of the right eye gaze point on the calibration plane, after calibration. The value is given in the Active Display Coordinate System.
`double`	`rightStatus`	Status code for the right eye.

GazeDataArray

The GazeDataArray represents a gaze data packet received from an eye tracker.

Type	Field	
`EyeArray`	`Left eye data`	An array containing the eye tracking data for the left eye for the current sample
`EyeArray`	`Right eye data`	An array containing the eye tracking data for the right eye for the current sample
`uint64`	`Eye tracker timestamp`	Timestamp for the point in time when the gaze data packet was sampled by the eye tracker and is given in the eye tracker clock time base. The value should be interpreted as a time interval in microseconds measured from an arbitrary point in time.
`uint32`	`trigSignal`	Reports whether the signal on the TX300's sync port is off or on: 0 = off, 1 = on.

EyeArray

The EyeArray contains the eye tracking data for the current sample.

Type	Field	
`double`	`EyePosition3d.x`	The X coordinate of the position of the eye in 3D space given in the User Coordinate System.
`double`	`EyePosition3d.y`	The Y coordinate of the position of the eye in 3D space given in the User Coordinate System.
`double`	`EyePosition3d.z`	The Z coordinate of the position of the eye in 3D space given in the User Coordinate System.
`double`	`EyePosition3dRelative.x`	The X coordinate of the position of the eye in 3D space relative to the track box given in the Track Box Coordinate System.
`double`	`EyePosition3dRelative.y`	The Y coordinate of the position of the eye in 3D space relative to the track box given in the Track Box Coordinate System.
`double`	`EyePosition3dRelative.z`	The Z coordinate of the position of the eye in 3D space relative to the track box given in the Track Box Coordinate System.
`double`	`GazePoint2d.x`	The X coordinate of the gaze point on the calibration plane given in the Active Display Coordinate System.
`double`	`GazePoint2d.y`	The Y coordinate of the gaze point on the calibration plane given in the Active Display Coordinate System.
`double`	`GazePoint3d.x`	The X coordinate of the gaze point on the calibration plane given in the User Coordinate System.
`double`	`GazePoint3d.y`	The Y coordinate of the gaze point on the calibration plane given in the User Coordinate System.
`double`	`GazePoint3d.z`	The Z coordinate of the gaze point on the calibration plane given in the User Coordinate System.
`double`	`PupilDiameter`	Diameter of the pupil in millimeters.
`double`	`Validity`	Validity code for the eye.

The Python language binding

Setting up the Python Environment

After installing the appropriate Python 2.7 variant from http://www.python.org/ you need to ensure that the Tobii SDK modules can be found on the Python path. For convenience, you might also want to add the path of python.exe to the path. The search path for Python modules is controlled by the environment variable PYTHONPATH (just like the search path for executables is controlled by PATH).

Specifying environmental variables in Windows

To set these environment variables, use the following procedure: If you unpacked the SDK zip to, e.g., `C:\Users\<name>\Desktop\[FolderName]` and Python is installed in `C:\Python27\` (which is the default location), open up a command prompt and enter the following commands:

- `set PATH=%PATH%;C:\Python27\`
- `set PYTHONPATH=C:\Users\<name>\Desktop\[FolderName]\Python27\Modules`

NOTE: If you use this method of setting the environment variables, the variables will only be valid for that command window or the child processes it starts. If the window is closed, the variables are no longer accessible. You can also set these environment variables permanently in the Windows System Properties.

Code samples

Once the environment variables are configured, the examples can be run from the command prompt by entering the following commands:

- `cd C:\Users\<name>\Desktop\[FolderName]\Python27\Samples`
- `python EyetrackerBrowser.py` (or any other example)

The subdirectory `Python27\Samples` contains python examples which use the PyGTK graphical toolkit. The PyGTK graphical toolkit is not installed by default.

For Windows PyGTK can be obtained from the following URL: http://www.pygtk.org/.

For Ubunty you may install it with `apt-get install python-gtk2`.

For Mac it should be installed by with `port install py27-pygtk -x11 +quartz`. It is important to get the quartz version of pygtk, since the X11 window system on Mac do not honor full screen requests in the correct way. Full screen is required for the calibration screen.

EyetrackerBrowser.py sample

If entered correctly, *EyetrackerBrowser.py* should launch a window, which looks like the image below (see Figure 21). This sample illustrates how to find eye trackers on the network, connect to a specific eye tracker, perform a calibration and subscribe to data from the eye tracker.

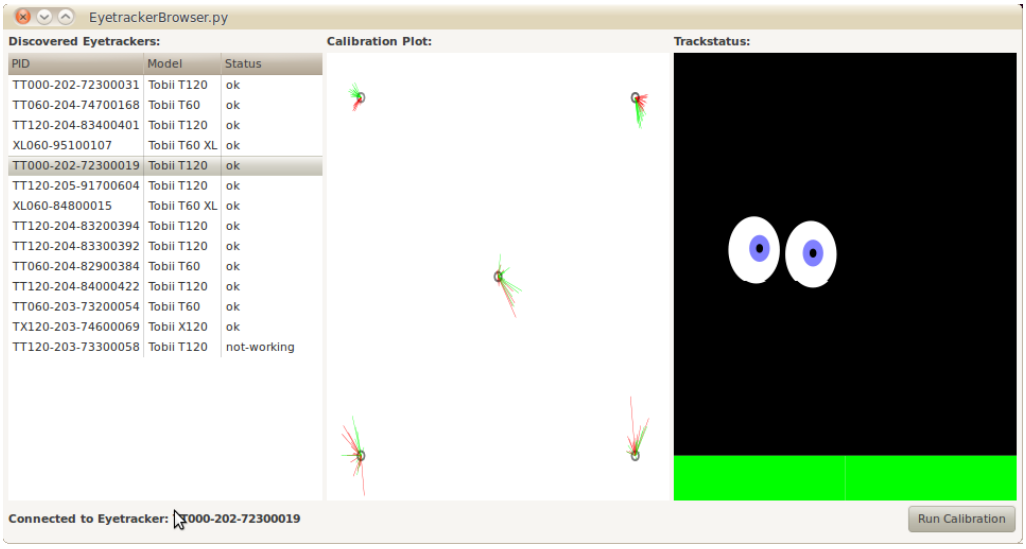


Figure 21. Python Eyetracker Browser example for PyGTK

UpgradeTool.py sample

The Python example collection also contains a small sample application that demonstrates the upgrade functionality provided by the SDK (see Figure 22). The sample can be found in the file UpgradeTool.py.

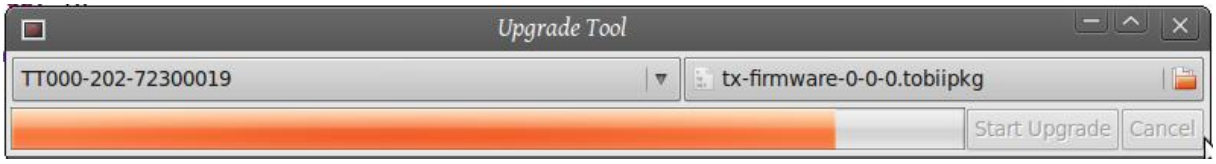


Figure 22. Python Upgrade Tool example application

Error handling

The SDK library contains an EyetrackerException that is available for the different languages. This exception contains an error code that further specifies what kind of error occurred. The different error codes and their interpretations are summarized in the section *Appendix 2. Error codes*.

Thread handling

Since the SDK must be able to handle data that arrives to the client in an asynchronous fashion, the core runs multiple threads that handle reads and writes to and from the eye tracker hardware. This has some implications for applications that use the SDK.

In the Python language bindings, all events are delivered on a background thread. This thread is provided by the user in the form of a MainloopThread object. The MainloopThread object is created by the client application and provided as an argument to all the constructors that need a mainloop instance. Typically, you create an instance of a MainloopThread and keep it alive for rest of the application's lifetime. When the application exits you must make sure to call MainloopThread.stop() to shut down all background threads.

Another important thing to keep in mind when developing Python applications is that you should never call back into the SDK from the background thread or deadlocks can occur. Instead you should direct all events from the background thread to the main thread. GUI frameworks typically offer this functionality since the user interface generally can be manipulated only from the main thread. In, e.g.,the Pygtk library, the glib_idle_add() function can be used to switch threads to the main thread. The Python sample applications show how to use the MainloopThread class and also how to marshal calls from background threads to the main UI thread.

Initializing the library

The Python language binding must be initialized before use. To do this, call the tobii.eye_tracking_io.init() method in your application startup code.

Class reference

Clock class

Namespace: tobii.eye_tracking_io.clock

The Clock class represents a high-resolution time source, which can be used for creating timestamps and/or for synchronizing clocks with an eye tracker.

Signature: class Clock (object):

Constructors	Description
`def __init__(self):`	Initializes the Clock class object.

Methods	Description
`get_resolution (self):`	Gets the clock resolution as a microsecond value.
`get_time (self):`	Gets the current time as a microsecond value.

Eyetracker class

Namespace: tobii.eye_tracking_io.eyetracker

The EyeTracker class represents an eye tracker.

Signature: class Eyetracker(tobii.eye_tracking_io.basic.BasicEyetracker):

The Eyetracker class exposes the following members:

Constructors	Description
`def __init__( self, message_passer):`	Initializes the Eyetracker object.

Methods	Description
`AddCalibrationPoint(self, point, callback=None, *args, **kwargs)`	Adds data to the temporary calibration buffer for one calibration point. The argument used is the point the calibrating user is assumed to be looking at and is given in the Active Display Coordinate System.
`ClearCalibration(self, callback=None, *args, **kwargs)`	Clears the temporary calibration buffer. This method should be called before starting a new calibration.
`ComputeCalibration(self, callback=None, *args, **kwargs)`	Uses the data in the temporary buffer and tries to compute calibration parameters. If the call is successful, data is copied from the temporary buffer to the active buffer. If there is insufficient data to compute a new calibration, or if the collected data is not good enough, then an exception will be thrown.
`EnumerateFramerates(self, callback=None, *args, **kwargs)`	Gets a list of possible frame rates for this eye tracker.
`EnumerateIlluminationModes(self, callback=None, *args, **kwargs)`	Gets a list of available illumination modes for this eye tracker. This method is only applicable for TX300 and T60XL eye trackers.
`GetCalibration(self, callback=None, *args, **kwargs)`	Gets a binary representation of the calibration. This data can be saved to file and used later.
`GetFramerate(self, callback=None, *args, **kwargs)`	Gets the current frame rate. A value of, e.g., 60.0 indicates that the current frame rate is 60 Hz.

`GetTrackBox(self, callback=None, *args, **kwargs)`	Gets the current track box. Returns a TrackBox instance containing eight 3D points given in the User Coordinate System.
`GetIlluminationMode(self, callback=None, *args, **kwargs)`	Gets the current illumination mode.
`GetLowblinkMode(self, callback=None, *args, **kwargs)`	Gets the current low-blink mode setting. This method is only applicable for T60/T120 and X60/X120 eye trackers.
`GetUnitInfo(self, callback=None, *args, **kwargs)`	Returns information about the eye tracker
`GetUnitName(self, callback=None, *args, **kwargs)`	Gets the eye tracker's name.
`GetXConfiguration(self, callback=None, *args, **kwargs)`	Gets the X configuration parameters.
`RemoveCalibrationPoint(self, point, callback=None, *args, **kwargs)`	Removes the data associated with a specific calibration point from the temporary calibration buffer.
`SetCalibration(self, calibration, callback=None, *args, **kwargs)`	Sets the provided calibration, which means copying the data from the temporary calibration buffer into the active calibration buffer.
`SetFramerate(self, framerate, callback=None, *args, **kwargs)`	Sets the current frame rate. The frame rate value must be one of the values returned from EnumerateFramerates or this method will fail.
`SetIlluminationMode(self, illuminationMode, callback=None, *args, **kwargs)`	Sets the illumination mode. This method is only applicable for TX300 and T60XL eye trackers.
`SetLowblinkMode(self, enabled, callback=None, *args, **kwargs)`	Sets the current low-blink mode setting. This method is only applicable for T60/T120 and X60/X120 eye trackers.
`SetUnitName(self, name, callback=None, *args, **kwargs)`	Sets the eye tracker's name.
`SetXConfiguration(self, UpperLeft, UpperRight, LowerLeft, callback=None, *args, **kwargs)`	Sets the X configuration parameters. This method will only work on x-units, e.g., the X120.
`StartCalibration(self, callback=None, *args, **kwargs)`	Acquires the calibration state and clears the temporary calibration buffer.
`StartTracking(self, callback=None, *args, **kwargs)`	Start subscribing to gaze data
`StopCalibration(self, callback=None, *args, **kwargs)`	Releases the calibration state. This should always be done when the calibration is finished!
`StopTracking(self, callback=None, *args, **kwargs)`	Stops gaze data subscription

`get_gaze_data_column(self, row, column_id)`	
`has_gaze_data_column(self, row, column_id)`	
`create_async(cls, mainloop, eyetracker_info, callback, *args, **kwargs)`	Creates a new Eyetracker instance from the supplied EyetrackerInfo. Attaches the eyetracker to the supplied mainloop, which can be either a Mainloop or MainloopThread object. The Eyetracker is delivered asynchronously to the supplied callback. callback is invoked as such: callback(eyetracker, *args, **kwargs)

Special Methods[9]	Description
`DumpImages(self, count, frequency, callback=None, *args, **kwargs)`	Method used to diagnose hardware problems.
`EnableExtension(self, extensionId, callback=None, *args, **kwargs)`	Enables a specific extension.
`GetAuthorizeChallenge(self, realmId, algorithms, callback=None, *args, **kwargs)`	Gets an authorization challenge to unlock custom functionality.
`GetAvailableExtensions(self, callback=None, *args, **kwargs)`	Gets a list of all available extensions.
`GetDiagnosticReport(self, include_images, callback=None, *args, **kwargs)`	Method used to diagnose hardware problems.
`GetEnabledExtensions(self, callback=None, *args, **kwargs)`	Gets a list of all enabled extensions.
`GetPayperuseInfo(self, callback=None, *args, **kwargs)`	Gets information whether this is a pay per use eye tracker or not.
`ValidateChallengeResponse(self, realmId, algorithm, responseData, callback=None, *args, **kwargs)`	Validates and unlocks custom functionality.

EyetrackerBrowser class

Namespace: tobii.eye_tracking_io.browsing

The EyeTrackerBrowser class is used to browse for available eye trackers on the network and/or connected through USB.

Signature: class EyetrackerBrowser(object):

[9] These methods are not recommended for use in normal application development. They are only included for the sake of completeness.

The EyetrackerBrowser class exposes the following members:

Constructors	Description
`def __init__(self, mainloop, callback, *args):`	Initializes the Eye tracker browser. Events from this browser will be posted on a mainloop thread. The arguments needed are a Mainloop or MainloopThread, the callback to use for events, and, optionally, arguments passed to the callback.

Methods	Description
`def stop(self):`	Stops the browsing for eye trackers.

EyetrackerEvents class

Namespace: tobii.eye_tracking_io.eyetracker

Provides the events used to signal errors or state changes within the SDK.

Signature: class EyetrackerEvents(tobii.eye_tracking_io.utils.events.Events):

The EyetrackerEvents class exposes the following members:

Constructors	Description
`def __init__(self, name):`	Initializes the EyeTracker event object.

Events	Description
`OnCalibrationStarted`	This event is fired when the calibration state has been reached.
`OnCalibrationStopped`	This event is fired when the calibration state has been left.
`OnFramerateChanged`	This event is fired when *another* client changes the frame rate.
`OnTrackBoxChanged`	This event is fired when the track box size has changed.
`OnXConfigurationChanged`	This event is fired when the X configuration has changed.
`OnGazeDataReceived`	This event is fired when a gaze data package has been received by the client application.
`OnError`	This event is fired when an error has been detected.

EyetrackerException class

Namespace: tobii.eye_tracking_io.basic

This is the type for exceptions thrown by the SDK.

Signature: class EyetrackerException(Exception):

The EyetrackerException class exposes the following members:

Constructors	Description
`def __init__(self, error):`	Creates an instance from a specific error code.

Properties	Description
`self.error`	Gets the error code value for this instance.

EyetrackerInfo class

Namespace: tobii.eye_tracking_io.browsing

The EyeTrackerInfo class describes an eye tracker found by browsing.

Signature: class EyetrackerInfo(object):

The EyetrackerInfo class exposes the following members:

Constructors	Description
`(None)`	It is not possible to create instances of the EyetrackerInfo class directly.

Properties	Description
`@property` `def product_id(self):`	Gets the product ID property as a string.
`@property` `def given_name (self):`	Gets the given name property as a string
`@property` `def model(self):`	Gets the model property as a string
`@property` `def generation(self):`	Gets the generation property as a string
`@property` `def firmware_version(self):`	Gets the version property as a string
`@property` `def status(self):`	Gets the status property as a string. This will be any of the values “ok”, ”not-working”, “upgrading” or “error”.
`@property` `def factory_info(self):`	Gets a FactoryInfo object from this EyetrackerInfo. This object is used when connecting to an eye tracker.

GazeDataItem class

Namespace: tobii.eye_tracking_io.types

Signature: class GazeDataItem(object):

The GazeDataItem class exposes the following members:

Constructors	Description
`def __init__(self):`	

Properties	Description
`Timestamp`	Timestamp for the point in time when the gaze data packet was sampled by the eye tracker and is given in the eye tracker clock time base. The value should be interpreted as a time interval in microseconds measured from an arbitrary point in time.
`LeftEyePosition3D`	The position of the left eye in 3D space given in the User Coordinate System.
`LeftEyePosition3DRelative`	The position of the left eye in 3D space relative to the track box given in the Track Box Coordinate System.
`LeftGazePoint2D`	The gaze point on the calibration plane for the left eye given in the Active Display Coordinate System.
`LeftGazePoint3D`	The gaze point on the calibration plane for the left eye given in the User Coordinate System.
`LeftPupil`	Diameter of the left pupil in millimeters.
`LeftValidity`	Validity code for the left eye.
`RightEyePosition3D`	The position of the right eye in 3D space given in the User Coordinate System.
`RightEyePosition3DRelative`	The position of the right eye in 3D space relative to the track box given in the Track Box Coordinate System.
`RightGazePoint2D`	The gaze point on the calibration plane for the right eye given in the Active Display Coordinate System.
`RightGazePoint3D`	The gaze point on the calibration plane for the right eye given in the User Coordinate System.
`RightPupil`	Diameter of the right pupil in millimeters.

`RightValidity`	Validity code for the right eye.
`TrigSignal`	(Optional) Signal sampled on the trig port of the Tobii TX-300 eye tracker.

Mainloop class

Namespace: tobii.eye_tracking_io.mainloop

The MainLoop class is responsible for running the mainloop within the SDK. The mainloop is a message pump/event dispatching mechanism.

Signature: class Mainloop(object):

The Mainloop class exposes the following members:

Constructors	Description
`def __init__(self):`	A mainloop is used by all asynchronous objects to defer handlers and callbacks to. The function run() blocks until the function quit() has been called (and all queued handlers have been executed). The run() function will then execute all the handlers in order.

Methods	Description
`def quit(self):`	Signals to the run() function that it should quit as soon as all deferred handles have been run.
`Def run(self):`	Executes deferred handles until quit() is called. WARNING: This function blocks until quit () is called.

MainloopThread class

Namespace: tobii.eye_tracking_io.mainloop

The MainloopThread class hosts the mainloop background thread for the application.

Signature: class MainloopThread(object):

The MainloopThread class exposes the following members:

Constructors	Description
`def __init__(self, mainloop=None, delay_start=false):`	Creates a new MainloopThread and either attaches an existing Mainloop to it or creates a new Mainloop. The argument delay_start (default: False) controls whether the thread should be started directly or not.

Methods	Description
`def start(self):`	Starts the mainloop thread. If the thread has already been started, this function does nothing.
`def stop(self):`	Stops the mainloop thread. If the thread is not currently running then this function does nothing.

State class

Namespace: tobii.eye_tracking_io.clock

The State class holds the current state of the time synchronization operation.

Signature: class State (object):

The State class exposes the following members:

Constructors	Description
`def __init__(self):`	Initializes the State class object.

Methods	Description
`def get_error_approximation (self):`	Returns an approximation of the current synchronization error.
`def points_in_use (self):`	Returns a list of tuples with three values in them, i.e. *local midpoint, remote timestamp*, *remote point timestamp* and *roundtrip time*.
`def state_flag(self):`	Describes the current state of the time synchronization operation which may be `UNSYNCHRONIZED, STABILIZING or SYNCHRONIZED`

SyncManager class

Namespace: tobii.eye_tracking_io.clock

The SyncManager class represents an ongoing time synchronization operation between the host computer and an eye tracker**.**

Signature: class SyncManager (object):

The SyncManager class exposes the following members:

Constructors	Description
`def __init__(self, clock, eyetracker_info, mainloop, error_handler=None, status_handler=None):`	Initializes the SyncManager class object with the arguments SDK clock and eye tracker ID.

Methods	Description
`def convert_from_local_to_remote(self, local_usecs):`	Converts from local to remote (eye tracker) time. The time unit is always microseconds.
`def convert_from_remote_to_local(self, remote_usecs):`	Converts from remote (eye tracker) time to local time. The time unit is always microseconds.

begin_upgrade function

Namespace: tobii.eye_tracking_io.upgrade

Signature:

Functions	Description
`begin_upgrade(mainloop, package_path, device_info, completed_handler, progress_handler, cancancel_handler)`	Parameters: - mainloop: either a Mainloop or MainloopThread - package_path: path to a tobiipkg file - device_info: an EyetrackerInfo - completed_handler: will be called like this: completed_handler(error_code) - progress_handler: will be called like this: progress_handler(current_step, number_of_steps, step_percentage)
`package_is_compatible_with_device(mainloop, package_path, device_info)`	

Appendices

Appendix 1. The Eye tracker browser and firmware upgrade

Verifying the current firmware version of the eye tracker

To verify the current firmware version of the eye tracker, start the Eye tracker browser application and select the eye tracker in the list of connected eye trackers. The eye tracker's firmware version can be seen in the information section about the eye tracker as illustrated in Figure 23. The three first sets of numbers describe the firmware version. In Figure 23 the eye tracker has firmware version 1.4.0.

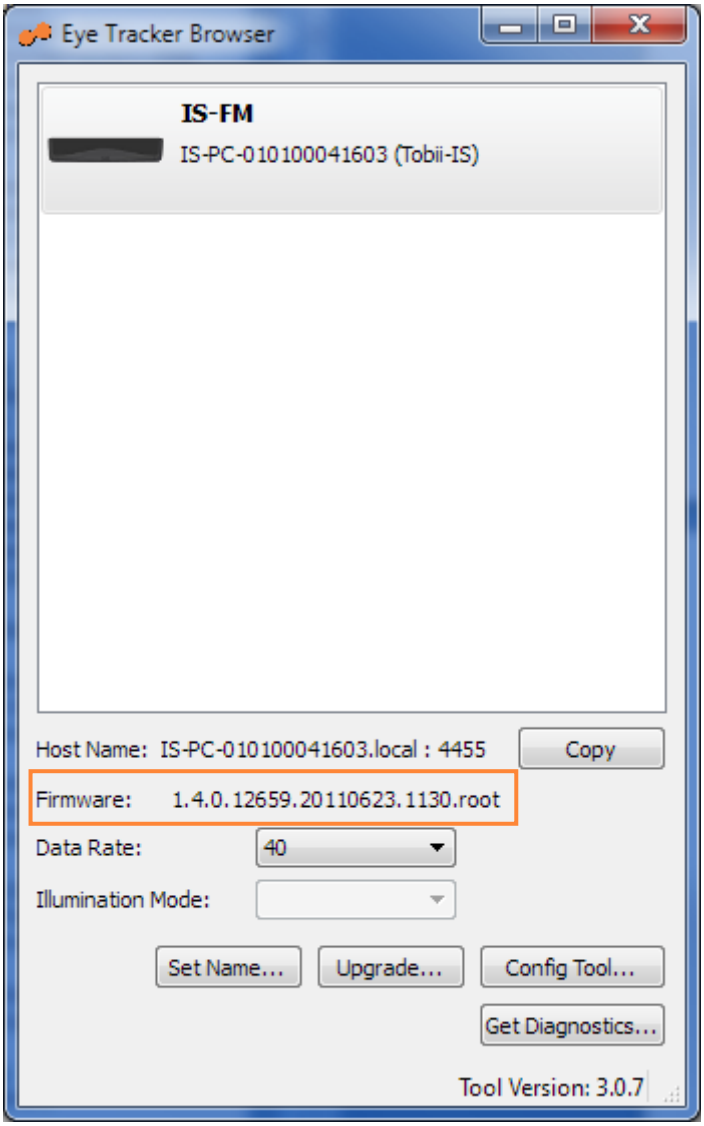


Figure 23. The Eye tracker browser application.

Firmware versions that support the Tobii SDK 3.0 are:

- Tobii X60, X120, T60 and T120: **2.0.0** *(or higher)*
- Tobii TX60 XL: **2.0.0** *(or higher)*
- Tobii TX300: **1.0.0** *(or higher)*
- Tobii X1 Light: **1.4.0.x.x.x** *(or higher)*

Upgrading the firmware

The first step in upgrading the firmware is to download it from the Tobii website. The newest firmware is listed in the list of downloadable files for each product. The numbers in the name of the firmware package also reveal the version of the firmware.

Once the firmware has been downloaded, open the Eye tracker browser, select the eye tracker to be upgraded in the list of available eye trackers and click the "Upgrade…" button (see Figure 24).

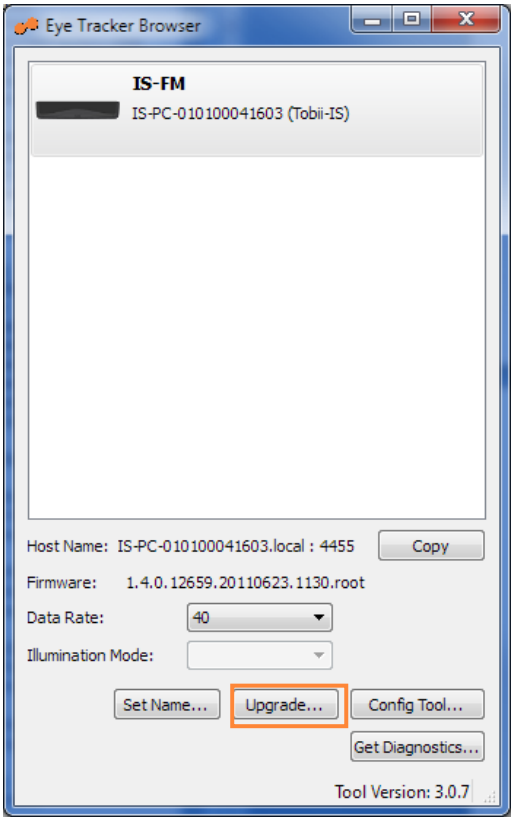


Figure 24. Eye Tracker Browser firmware update button.

The next step is to locate the downloaded firmware package file and open it. The file should have the ending ***.tobiipkg** (see Figure 25).

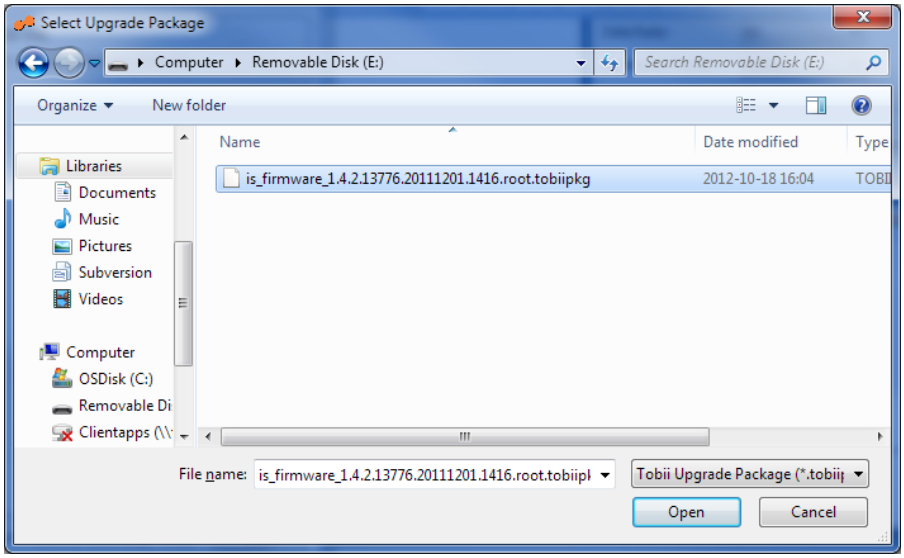


Figure 25. Locating the firmware package.

Once the file has been opened, the firmware upgrade will be made automatically. You can follow the progress of the upgrade in the upgrade progress windows as seen in Figure 26. Once the upgrade is completed, click "OK" to finish the process.

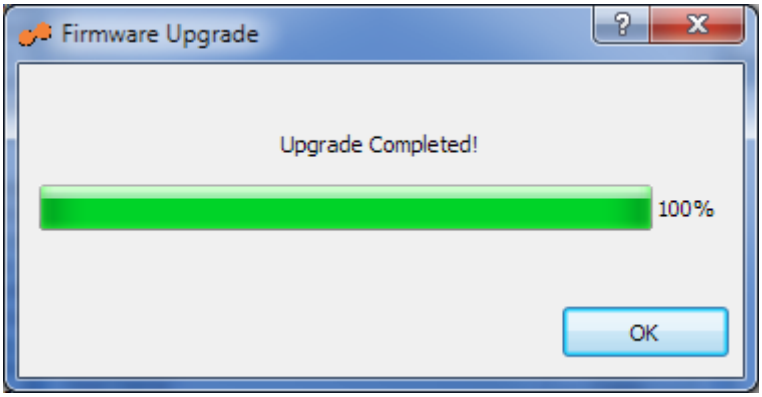


Figure 26. Upgrade progress window.

If the upgrade progress is not successful during the first attempt, repeat the process. If the process still fails, check that the firmware package is for the selected eye tracker. If the eye tracker is working correctly with the old firmware but the update still fails, contact Tobii support.

Appendix 2. Error codes

The following tables list all currently defined error codes in the firmware and in the current SDK. Many of the error codes listed here are internal and are listed for your reference.

Generic Errors

Name	Code (Decimal)	Code (Hex)	Description
`TOBII_SDK_ERROR_SUCCESS`	0	0	No error
`TOBII_SDK_ERROR_GENERIC`	1	1	Represents a generic error, i.e. no other suitable error exists
`TOBII_SDK_ERROR_INVALID_ARGUMENTS`	2	2	One or several of the arguments to a method/function were invalid. Typically occurs when one of the parameters is null.
`TOBII_SDK_ERROR_OUT_OF_MEMORY`	3	3	Not enough memory to perform a specific operation. Typically a severe error condition that is hard to handle.
`TOBII_SDK_ERROR_OUT_OF_RANGE`	4	4	One or several function/method arguments were out of range.
`TOBII_SDK_ERROR_NOT_INITIALIZED`	5	5	The SDK has not been initialized.
`TOBII_SDK_ERROR_NOT_SUPPORTED`	6	6	The called function is not supported.
`TOBII_SDK_ERROR_TIMEOUT`	30	0x0000001E	An outstanding sync operation timed out. Can occur when connection to the eye tracker is lost.
`TOBII_SDK_ERROR_OPERATION_ABORTED`	32	0x0000001F	An outstanding sync operation was aborted before it had time to complete.

Param Stack Errors

Name	Code (Decimal)	Code (Hex)	Description
`TOBII_SDK_ERROR_INVALID_PAYLOAD_ITEM_TYPE`	7	7	Occurs when the SDK application tries to serialize data of an unknown type.

Transport Errors

Name	Code (Decimal)	Code (Hex)	Description
TOBII_SDK_ERROR_TRANSPORT_ERROR	8	8	Internal Error
TOBII_SDK_ERROR_UNKNOWN_OPCODE	9	9	Internal Error
TOBII_SDK_ERROR_INVALID_PAYLOAD	10	0x0000000A	Internal Error. Message sent from the eye tracker contains invalid data
TOBII_SDK_ERROR_UNEXPECTED_PAYLOAD	11	0x0000000B	Internal Error. Message sent from the eye tracker contained data but none was expected
TOBII_SDK_ERROR_EMPTY_PAYLOAD	12	0x0000000C	Internal Error. Message sent from the eye tracker did not contain any data but data was expected.

Factory Info Errors

Name	Code (Decimal)	Code (Hex)	Description
TOBII_SDK_ERROR_INVALID_FACTORYINFO	20	0x00000014	Internal Error

Firmware Upgrade Errors

Name	Code (Decimal)	Code (Hex)	Description
TOBII_SDK_ERROR_UPGRADE_GENERIC	40	0x00000028	A generic error occurred during upgrade.
TOBII_SDK_ERROR_UPGRADE_SESSION_MISMATCH	41	0x00000029	Internal Error
TOBII_SDK_ERROR_UPGRADE_MISSING_PART_ID	42	0x0000002A	The upgrade package contains data that the SDK cannot understand.
TOBII_SDK_ERROR_UPGRADE_PACKAGE_VALIDATION	43	0x0000002B	The selected upgrade package file is not compatible with the selected eye tracker.

TOBII_SDK_ERROR_UPGRADE_WRONG_MODEL	44	0x0000002C	The selected upgrade package file is not compatible with the selected eye tracker. The package was made for another model.
TOBII_SDK_ERROR_UPGRADE_WRONG_GENERATION	45	0x0000002D	The selected upgrade package file is not compatible with the selected eye tracker. The package was made for another product generation.
TOBII_SDK_ERROR_UPGRADE_CANT_DOWNGRADE	46	0x0000002E	The installed firmware is newer than the firmware contained in the upgrade package file and downgrading is not supported.
TOBII_SDK_ERROR_UPGRADE_DEVICE_DATA_MISSING	47	0x0000002F	Failed to read data from the eye tracker when running compatibility checks.
TOBII_SDK_ERROR_OPERATION_ABORTED	32	0x0000001F	The firmware upgrade operation was aborted.

Eye Tracker Errors

These are errors that are returned by the eye tracker in response to a request from the client.

Name	Code (Decimal)	Code (Hex)	Description
TOBII_FW_ERROR_SUCCESS	0	0	No error
TOBII_FW_ERROR_UNKNOWN_OPERATION	536872192	0x20000500	The specific opcode or request is unknown. This means that the eye tracker doesn't understand what it is supposed to do with this request.
TOBII_FW_ERROR_UNSUPPORTED_OPERATION	536872193	0x20000501	The eye tracker understands the opcode or request, but does not support it.
TOBII_FW_ERROR_OPERATION_FAILED	536872194	0x20000502	The request failed. This can mean different things for different requests.

TOBII_FW_ERROR_INVALID_PAYLOAD	536872195	0x20000503	The request contained invalid data. This is a severe error and indicates some kind of network problem.
TOBII_FW_ERROR_UNKNOWN_ID	536872196	0x20000504	The opcode referenced an unknown ID. This is probably caused by a programming error.
TOBII_FW_ERROR_UNAUTHORIZED	536872197	0x20000505	The operation cannot be completed without further authorization.
TOBII_FW_ERROR_EXTENSION_REQUIRED	536872198	0x20000506	The operation requires enabling an extension.
TOBII_FW_ERROR_INTERNAL_ERROR	536872190	0x20000507	A generic error that indicates a bug in the firmware.
TOBII_FW_ERROR_STATE_ERROR	536872200	0x20000508	The server or client is in a state where this request is unsupported or not allowed. For example, this can occur when trying to call AddCalibrationPoint without first calling StartCalibration()
TOBII_FW_ERROR_INVALID_PARAMETER	536872201	0x20000509	One or more parameters in the request were incorrect.
TOBII_FW_ERROR_OPERATION_ABORTED	536872202	0x20000510	The request was aborted before it could complete. This indicates that the network connection to the eye tracker was lost.

Appendix 3. Tobii T, X, and TX eye tracker data reference

Field	Type	Description
Timestamp	8 byte signed integer	A microsecond timestamp from the eye tracker's internal clock.
Left Eye Position	3D point given as three 8 byte floating point numbers (x, y and z)	The 3D position of the left eye in relation to the eye tracker. User Coordinate System.
Right Eye Position	3D point given as three 8 byte floating point numbers (x, y and z)	The 3D position of the right eye in relation to the eye tracker. User Coordinate System.
Left Eye Relative Position	3D point given as three 8 byte floating point numbers (x, y and z)	The 3D position of the left eye relative to the Track Box Coordinate System.
Right Eye Relative Position	3D point given as three 8 byte floating point numbers (x, y and z)	The 3D position of the right eye relative to the Track Box Coordinate System.
Left Eye Gaze Point	3D point given as three 8 byte floating point numbers (x, y and z)	The 3D position of the left eye gaze point on the calibration plane. User Coordinate System.
Right Eye Gaze Point	3D point given as three 8 byte floating point numbers (x, y and z)	The 3D position of the right eye gaze point on the calibration plane. User Coordinate System.
Left Eye Relative Gaze Point	2D point given as two 8 byte floating point numbers (x, y)	The 2D position of the left eye gaze point in the Active Display Coordinate System.
Right Eye Relative Gaze Point	2D point given as two 8 byte floating point numbers (x, y)	The 2D position of the right eye gaze point in the Active Display Coordinate System.
Left Eye Validity Code	4 byte unsigned integer	A value describing if the left eye was found.
Right Eye Validity Code	4 byte unsigned integer	A value describing if the right eye was found.
Left Eye Pupil Diameter	8 byte floating point number	The pupil diameter of the left eye in millimeters.
Right Eye Pupil Diameter	8 byte floating point number	The pupil diameter of the right eye in millimeters.