AdHawk SDK

Version v5.10

August 16, 2022

STRICTLY CONFIDENTIAL

AdHawk Microsystems

410 Albert St, unit 102, Waterloo, ON, N2L 3V3, Canada

info@adhawkmicrosystems.com

Contents

Overview **5**
Communication Protocol . . . 6
Control Interface . . . 6
Stream Interface . . . 6
Backend Service . . . 6

Getting Started **7**
Installation . . . 7
Python SDK . . . 7
C SDK . . . 7
Setting up Communication . . . 7
Auto-Tune . . . 8
Using backend built-in calibration procedure with GUI . . . 9
Enable data streams . . . 9
Saving Calibrations . . . 10
Gaze In Image . . . 10
Screen Tracking . . . 12

API Reference **14**
Control Packets . . . 14
Calibration start [0x81] . . . 14
Calibration complete [0x82] . . . 14
Calibration abort [0x83] . . . 15
Register calibration point [0x84] . . . 15
Trigger Autotune [0x85] . . . 16
Re-center calibration [0x8f] . . . 16
Get tracker status [0x90] . . . 17
Start camera [0xd2] . . . 17
Stop camera [0xd3] . . . 17
Subscribe for video stream [0xd4] . . . 18
Unsubscribe from video stream [0xd5] . . . 18
Register Screen Aruco Board [0xda] . . . 18
Screen tracking start [0xdb] . . . 19
Screen tracking stop [0xdc] . . . 19
Run-time configuration objects (blobs) . . . 20
Blob size [0x92] . . . 20
Blob data [0x93] . . . 21
Load blob [0x94] . . . 22
Save blob [0x95] . . . 22
Properties . . . 23
Get autotune position [0x9a, 0x01] . . . 23
Set autotune position [0x9b, 0x01] . . . 23

Set stream control [0x9b, 0x02] . . . 24
Get stream control [0x9a, 0x02] . . . 24
Get component offsets [0x9a, 0x04] . . . 25
Set component offsets [0x9b, 0x04] . . . 25
Set event control [0x9b, 0x05] . . . 26
Get event control [0x9a, 0x05] . . . 26
Get normalized eye offsets [0x9a, 0x09] . . . 26
System Control . . . 27
Enable / Disable Eye Tracking [0x9c, 0x01] . . . 27
Device Setup Procedures . . . 27
Trigger Device Calibration [0xb0, 0x01] . . . 27
Get Device Calibration Status [0xb1, 0x01] . . . 27
Trigger Update Firmware [0xb0, 0x02] . . . 28
Get Update Firmware Status [0xb1, 0x02] . . . 28
Marker Sequence Procedures . . . 29
Calibration GUI Start [0xb0, 0x03] . . . 29
Get Calibration GUI Status [0xb1, 0x03] . . . 29
Validation GUI Start [0xb0, 0x04] . . . 29
Get Validation GUI Status [0xb1, 0x04] . . . 30
Autotune GUI Start [0xb0, 0x05] . . . 30
Get Autotune GUI Status [0xb1, 0x05] . . . 30
Quick-start GUI Start [0xb0, 0x06] . . . 30
Get Quick-start GUI Status [0xb1, 0x06] . . . 31
Camera user settings . . . 31
Set gaze depth [0xd0, 0x01] . . . 31
Enable/Disable parallax correction [0xd0, 0x02] . . . 31
Set sampling duration [0xd0, 0x03] . . . 32
Backend service communication . . . 32
Register endpoint [0xc0] . . . 32
Deregister endpoint [0xc2] . . . 33
Ping endpoint [0xc5] . . . 33
Data Streams . . . 33
Tracker ready [0x02] . . . 33
Gaze vector stream [0x03] . . . 34
Pupil position stream [0x04] . . . 34
Pupil diameter stream [0x05] . . . 35
Per-eye gaze vector stream [0x06] . . . 36
Gaze in image stream [0x07] . . . 37
Gaze in screen stream [0x08] . . . 38
IMU data stream [0x17] . . . 39
IMU Rotation data stream [0x19] . . . 39
Events . . . 40
Blink events . . . 40
Trackloss Events . . . 41
Types reference . . . 41
Return codes . . . 41
Stream Control Bitmask . . . 43
Event Control Bitmask . . . 43
Blob Type . . . 43
Supported Rates . . . 43
Camera Resolution . . . 44
Procedure Types . . . 44
Marker Sequence Mode . . . 45
Camera User Setting Types . . . 45

Extended Usage Notes **46**
Python SDK 46
Synchronous vs Aysnchronous operation 46

AdHawk Coordinate System **47**
Image Coordinate System 48
Screen Coordinate System 48

Changelog **49**
v5.8 49
v5.9 49

Overview

The purpose of this document is to describe the APIs used to interface with AdHawk's Eye Tracking module.

Communication with the AdHawk Eye Tracking module can be accomplished via:

1. AdHawk API Communication Protocol
2. C SDK, which is built on top of the communication protocol and communicates with the device directly over USB or SPI.

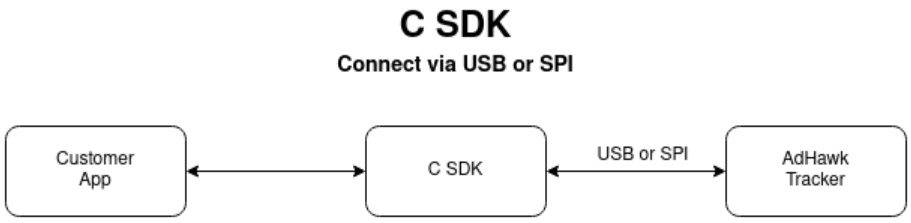

Figure 1: C SDK via USB or SPI

3. Python SDK, which is built on top of the communication protocol, but proxies commands through the AdHawk Backend Service. The C SDK can also be set up to proxy commands in this fashion. The advantage of running in this mode is that multiple apps can be run at the same time.

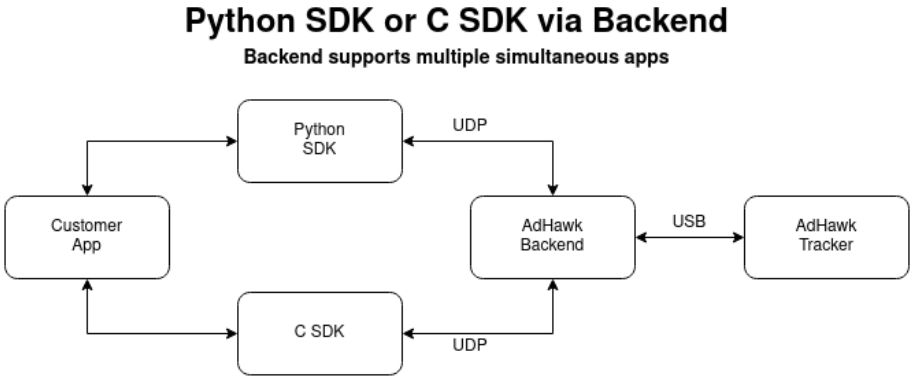

Figure 2: Python or C SDK via Backend

Communication Protocol

The communication packets consist of a packet type followed by zero or more data bytes:

Byte	Type	Description
0	uint8	The packet type
1...	...	Payload

- The length of payload is variable depending on the packet type
- All values are stored in little endian (least significant byte first)

Control Interface

The control interface provides the ability to interact with AdHawk's Eye Tracking module. The client/host initiates a command over the control interface by sending a request packet. The Control Packets section describes the list of available commands that the client/host can request over the control interface.

A request to the control interface may take up to 8 seconds to complete. All requests to the control interface will receive a response, either containing the requested information, or containing the result of the request. The clients should only issue one request at a time.

The responses that contain the result of the request consist of a 1-byte integer, where 0 indicates no error and successful execution of the requested command, and a non-zero value denotes the nature of error encountered while servicing the request. See the return codes section for details.

Stream Interface

AdHawk's Eye Tracking module streams status and tracking data. Some streams like the tracker ready signal is always broadcasted. Other streams, such as the gaze vector stream, must be subscribed to.

Backend Service

The AdHawk Backend Service is the PC host component that communicates with AdHawk's eye tracking hardware over USB, and allows multiple clients to interact with and receive data from the eye tracking module. In order to support multiple clients simultaneously, all communications between clients and the Backend Service is performed using UDP sockets. There are two sockets: control and data.

The control socket is constant, and is configured to 11032. The backend service is always listening on the control socket for client requests, and the responses to those requests are sent back to the source UDP port that was specified in the request packet.

The data socket is used to stream periodic eye tracking data to clients. Each client specifies its data port by sending a register endpoint packet to the control socket. The client data port may be the same as its control data port if desired.

Getting Started

Installation

Python SDK

```
pip install adhawk_py_sdk-<version>.tar.gz
```

C SDK

Refer to the README.md in the C SDK package

Setting up Communication

If communicating with the AdHawk Backend Service, a communication channel must be established over UDP. Start by registering as an endpoint by sending the register endpoint packet with the UDP port of the peer.

If using the Python SDK, this is handled by the FrontendApi and can be accomplished as follows:

```
import adhawkapi
import adhawkapi.frontend

def on_connect(error):
    if not error:
        print('Connected to AdHawk Backend Service')

def on_disconnect(error):
    print('Disconnected from AdHawk Backend Service')

api = adhawkapi.frontend.FrontendApi()
api.start(connect_cb=on_connect, disconnect_cb=on_disconnect)
...
# Terminate communication
api.shutdown()
```

The C SDK only requires the AdHawk Backend Service when running in UDP mode. The USB and SPI modes communicate directly with the device and do not require the AdHawk Backend Service. The API initialization call will take care of all setup required depending on the specified mode. See `ah_com.h` for details on how to connect in different modes

```
ah_api_init(params)
```

Auto-Tune

To tune the device, send the trigger autotune packet while the user is looking straight ahead. The response indicates whether the Auto-Tune procedure was successful.

C

```
ah_result ah_api_runAutotune(ah_autotuneReferenceGazeVector *referenceGazeVec);
```

Python

```
api.trigger_autotune(reference_gaze_vector)
```

The caller can optionally provide a reference gaze vector to trigger autotune for more accurate results. The reference gaze vector is a 3-dimensional vector representing the point in world space the user is looking at during the tuning procedure. It consists of an X, Y, and Z component (all values are expected to be in meters). ## Calibrate

Once the tracker ready signal is received or get tracker status returns success, the device is ready to be calibrated.

Calibration is the process of mapping the eye tracking signal to an individual's visual axis, which makes the eye tracking output meaningful for typical applications. To initiate the calibration procedure send a calibration start packet.

The calibration procedure consists of the collection of a set of X, Y, Z gaze references. The gaze references are collected by instructing the user to look at a prescribed point and signaling the registration of the calibration data point through a register calibration point packet. When registering a calibration data point, the client must instruct AdHawk's eye-tracking module of the user's expected gaze during the collection of the calibration point. The expected gaze is specified in the payload of the Register calibration data point packet. Note that a response of 'Left Eye Not Found' or 'Right Eye Not Found' can be treated as a warning in binocular operation, but 'Eyes Not Found' is an error.

After collecting the data points, send the calibration complete packet to indicate that no further points will be collected and that a new calibration should be generated.

Once the calibration complete packet is received the eye-tracking module will compute the calibration coefficients based on the correspondence of sensor data and gaze references. The AdHawk's eye-tracking module will respond with a zero (0) return code for successful calibration, and a non-zero value if calibration fails. See the return codes section for details.

If calibration is successful, the device is capable of streaming the gaze vector stream and the pupil position stream.

C

```
ah_api_calibrationStart();
for (unsigned i = 0; i < num_points; i++)
{
    float x, y, z;
    // As the eye moves and fixates on each point on the screen,
    // register the position of the point
    // get_reference_point(&x, &y, &z);
    ah_api_calibrationRegisterPoint(x, y, z);
}
ah_api_calibrationComplete();
```

Python

```
def grid_points(nrows, ncols, xrange=20, yrange=12.5, xoffset=0, yoffset=2.5):
    '''Generates a grid of points based on range and number of rows/cols'''
```

```
    zpos = -0.6  # typically 60cm to screen
    # calculate x and y range for a xrange x yrange degree calibration window
    xmin, xmax = np.tan(np.deg2rad([xoffset - xrange, xoffset + xrange])) * np.abs(zpos)
    ymin, ymax = np.tan(np.deg2rad([yoffset - yrange, yoffset + yrange])) * np.abs(zpos)

    cal_points = []
    for ypos in np.linspace(ymin, ymax, nrows):
        for xpos in np.linspace(xmin, xmax, ncols):
            cal_points.append((xpos, ypos, zpos))

    print(f'grid_points(): generated {nrows}x{ncols} points'
          f' {xrange}x{yrange} deg box at {zpos}: {cal_points}')

    return cal_points

npoints = 9
nrows = int(np.sqrt(npts))
reference_points = grid_points(nrows, nrows)

api.start_calibration()
for point in reference_points:
    # As the eye moves and fixates on each point on the screen,
    # register the position of the point
    api.register_calibration_point(*point)
api.stop_calibration()
```

Using backend built-in calibration procedure with GUI

> Only supported when using the AdHawk Backend Service
>
> Currently not supported in c sdk

The calibration of gaze tracking can also be done with a graphical user interface provided by the AdHawk Backend Service that guides the user through the procedure. This API launches a built-in calibration screen provided by the AdHawk Backend service. The user needs to fixate at the center of a sequence of targets displayed on the screen and register the calibration samples. The GUI will be closed automatically at the end of the calibration. See Marker Sequence Procedures for more details about the packet.

Python

```
api.start_calibration_gui(mode=adhawkapi.MarkerSequenceMode.FIXED_GAZE,
                          n_points=5, marker_size_mm=35,
                          randomize=True,
                          fov=(30, 25, 0, -10),
                          callback=lambda *x: None)
```

Enable data streams

Once the device is calibrated, to start receiving eye tracking data from the device, send a set stream control packet with the stream you want to enable and the rate at which you want to enable it. See Supported rates.

At this point, the device will start streaming packets to the registered endpoint. The format of the streamed packets are listed in the Data Streams section.

C

```c
static void gazeDataCallback(void *data)
{
    ah_gazeStreamData *gaze = data;
}

ah_api_setupStream(ah_streamType_Gaze, ah_streamRate_60Hz, gazeDataCallback);
```

Python

```python
def handler(*data):
    timestamp, xpos, ypos, zpos, vergence = data

api.register_stream_handler(adhawkapi.PacketType.GAZE, gaze_handler)
api.set_stream_control(adhawkapi.PacketType.GAZE, 60)
```

Saving Calibrations

Once a device has been calibrated, the calibration coefficients can be saved using the blobs framework. The blobs framework allows saving and restoring run-time configuration objects of the eye-tracking module.

Send a save blob packet with the blob type set to 1 (gaze calibration data). If the blob was saved successfully, a unique 32-bit identifier is returned in the response packet. The calibration can later be loaded by sending a load blob packet with this ID in the payload.

C

```c
ah_api_calibrationSave();
```

Python > Only supported when using the AdHawk Backend Service

```python
errcode, blob_id = api.save_blob(adhawkapi.BlobType.CALIBRATION)
```

Gaze In Image

Currently not supported in c sdk

While the main output of the gaze tracker is a stream of user's gaze ray in 3d, the frontend can request a stream of (x, y) values as the projection of the gaze ray inside the image captured by the front-facing camera. This also assumes that the gaze tracker is already calibrated for the user.

To be able to see the gaze projection in the camera frame in realtime, you need to first start the camera. Once the camera is started and a success response code is received you can subscribe for the video stream. Starting the camera is needed for calculating the gaze projection based on the chosen image resolution, however you don't necessarily need to subscribe for the video stream. Receiving the video stream only allows you to draw the image and overlay the gaze point on top of it.

In order to get the gaze-in-image stream you need to subscribe for the stream and enable that stream. Note that you don't need to subscribe for the gaze vector for this but the rate you will receive the gaze-in-image stream is the same rate that is set for the gaze vector stream.

Python

```python
import adhawkapi.frontend

def handle_connect(error):
    if not error:
        # trigger camera start
        api.start_camera_capture(camera_device_index, resolution_index, undistort_image, callback=handl
```

```
        # register for gaze-in-image stream
        api.set_stream_control(api.PacketType.GAZE_IN_IMAGE, 1, callback=(lambda *args: None))

def handle_camera_start_response(response):
    if response:
        print(f'camera start response: {response}')
    else:
        start_video_stream()

def start_video_stream():
    ''' Start the video streaming '''
    api.start_video_stream(*video_receiver.address, lambda *x: None)

def frame_handler(timestamp, frame_index, image_buf):
    '''
    display the image and overlay the gaze_xy point

    example decoder 1: decode the image using opencv
        image = cv2.imdecode(np.frombuffer(image_buf, dtype=np.uint8), 1)

    example decoder 2: decode the image in pyqtgraph
        qt_img = QtGui.QPixmap()
        qt_img.loadFromData(image_buf, 'JPEG')

    '''
    return

def handle_gaze_in_image_data(_timestamp, gaze_img_x, gaze_img_y, _deg_to_pix_x, _deg_to_pix_y):
    ''' Simple handler that prints the gaze coordinates '''
    global gaze_xy
    gaze_xy = [gaze_img_x, gaze_img_y]
    print(f'gaze coordinates in the image: {gaze_xy}')

def shutdown():
    api.stop_video_stream(*video_receiver.address, lambda *x: None)
    api.stop_camera_capture(lambda *_args: None)
    api.shutdown()

gaze_xy = (0, 0)

# setup a video receiver
video_receiver = adhawkapi.frontend.VideoReceiver(frame_handler)
video_receiver.start()

# define the camera properties
camera_device_index = 0
resolution_index = 0
undistort_image = False

# start
api = adhawkapi.frontend.FrontendApi()
api.register_stream_handler(adhawkapi.PacketType.GAZE_IN_IMAGE, handle_gaze_in_image_data)
api.start(connect_cb=handle_connect)
```

```
# shutdown when you are done
shutdown()
```

Screen Tracking

Currently not supported in c sdk

The Adhawk head-mounted eye tracker estimates the user's gaze in 3d (as well as the projection in the image), whereas majority of the table-mounted eye trackers estimate the gaze as a point inside the screen coordinate system. In order to project the gaze data estimated relative to the user's head in 3d, onto the screen plane, the screen has to be located and tracked relative to the user's head (a.k.a. screen tracking). This is done by tracking the screen in the image captured by the front-facing camera. While various methods can be used for tracking the screen, the Adhawk eye tracking module does this by tracking one or multiple Aruco markers displayed in the screen plane. This is done via OpenCV and taking advantage of the IMU sensor integrated on the Adhawk eye tracking glasses to track the head movements.

The first step you need to do before using the screen tracking feature is to prepare the screen setup as follows:

- Take a set of aruco markers from an aruco dictionary and display them on the same plane as the screen. The markers can have different sizes, can be physical (printed on a piece of paper and glued to a cardboard to make it flat and rigid) or virtual (generated in the software and shown on a screen). The markers can be placed anywhere inside or outside the screen area. Make sure there is a white margin around each marker at least as big as the black border of the marker.
- Make sure that all the markers are flat and oriented correctly in 3d such that all the 4 corners are placed in an imaginary plane that passes through the screen plane.
- Measure the bottom-left corner of each marker relative to the origin Screen Coordinate System (top-left corner of the screen) in meters.

Figure below shows an example with 4 markers and the measurement to take.

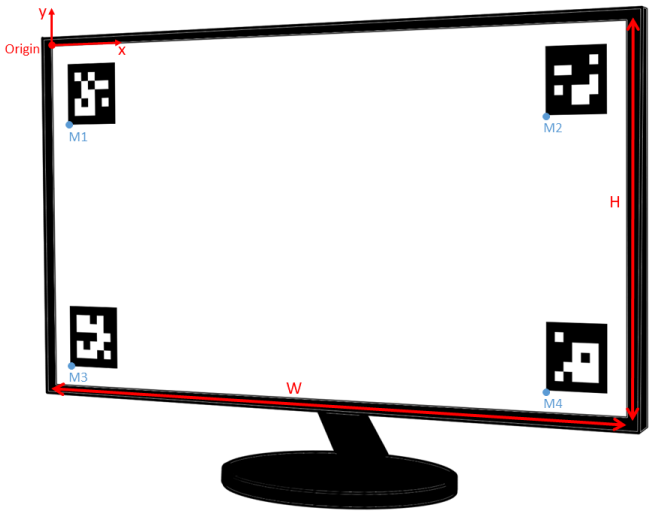

Figure 3: Screen Coordinate System

In order to get the gaze-in-screen data, you need to register the aruco markers. Registering the screen markers can be done by sending a register screen command that defines your screen size, aruco dictionary, index of each marker (as defined in the dictionary), the position of the bottom-left corner of each maker in the screen coordinate system, as well as the size of each marker in meters. Multiple markers can be added to the same packet. After the registering the screen, enable screen tracking, and subscribe to the

gaze-in-screen stream. The Adhawk eye tracking module will stream the estimated gaze position inside your screen coordinate system as (timestamp, x, y) values where the timestamp matches the timestamp of the corresponding gaze vector stream and the (x, y) are normalized float values (0-1) where (0,0) is the top-left corner and (1, 1) is the bottom-right corner of the screen.

Python

```
import cv2
import adhawkapi

aruco_dictionary = cv2.aruco.DICT_5X5_50
sceen_size = (0.345, 0.195)  # in meters
marker_size = 0.03  # in meters
marker_ids = [0, 1, 2, 3]
markers_pos = [[0.0, 0.12], [0.27, 0.12], [0.0, 0.0], [0.27, 0.0]]

api.register_screen_board(*sceen_size, aruco_dictionary, marker_ids,
                          [[pos[0], pos[1], marker_size] for pos in markers_pos])
api.start_screen_tracking()
def handler(*data):
    timestamp, xpos, ypos = data

api.register_stream_handler(adhawkapi.PacketType.GAZE_IN_SCREEN, handler)
```

API Reference

Control Packets

Calibration start [0x81]

A calibration start command packet will instruct AdHawk's eye-tracking module to reset the calibration table and prepare for a new calibration procedure.

API

	Byte	Type	Description
Request	0	uint8	0x81
Response	0	uint8	0x81
	1	uint8	Return code

C

```
ah_result ah_api_calibrationStart(void);
```

Python

```
api.start_calibration()
```

Calibration complete [0x82]

A calibration complete command will instruct AdHawk's eye-tracking module that no further points will be collected and that a new calibration should be generated. Once the Calibration complete packet is received the eye-tracking module will compute the calibration coefficients based on the correspondence of sensor data and gaze references. The AdHawk's eye-tracking module will respond with a zero (0) return code for successful calibration, and a non-zero value if calibration fails.

API

	Byte	Type	Description
Request	0	uint8	0x82
Response	0	uint8	0x82
	1	uint8	Return code

C

```
ah_result ah_api_calibrationComplete(void);
```

Python

```
api.stop_calibration()
```

Calibration abort [0x83]

A calibration abort packet will instruct AdHawk's eye-tracking module to discard all collected data for the new calibration, and restores the previous calibration.

API

	Byte	Type	Description
Request	0	uint8	0x83
Response	0	uint8	0x83
	1	uint8	Return code

C

```
ah_result ah_api_calibrationAbort(void);
```

Python

```
api.abort_calibration()
```

Register calibration point [0x84]

Once a target gaze point is established and the user is instructed to direct their gaze at the target, a calibration data point may be registered. Registering a data point requires the client to instruct the eye-tracking module of the intended gaze target such that a correspondence can be established between the user's gaze and the sensor data.

AdHawk's eye-tracking module will respond with a zero (0) return code if the calibration point is successfully registered. Otherwise, a non-zero payload will be returned indicating that a correspondence between the user's gaze and sensor data could not be established.

X, Y and Z is the position of the point relative to the midpoint of the scanners. See AdHawk Coordinate System for details.

API

	Byte	Type	Description
Request	0	uint8	0x84
	1-4	float32	X position in meters
	5-8	float32	Y position in meters
	9-12	float32	Z position in meters
Response	0	uint8	0x84
	1	uint8	Return code

C

```
ah_result ah_api_calibrationRegisterPoint(float xPos, float yPos, float zPos);
```

Python

```
api.register_calibration_point(0.0, -0.179, -0.6)
```

Trigger Autotune [0x85]

When trigger scanner ranging is signaled, the eye-tracking module will perform an auto-tune function on all attached trackers. The eye-tracking module will respond with a zero (0) to indicate that scanner ranging successfully completed on both eyes. Otherwise, a non-zero payload will indicate which eye encountered a failure.

The endpoint also supports an optional reference gaze vector within the payload. This reference gaze vector is an estimate of the world space position (in meters) that the user is looking at during the tuning process. Providing a reference gaze vector will increase the overall accuracy of the tuning process. This reference should be supplied whenever possible.

API

	Byte	Type	Description
Request	0	uint8	0x85
Optional Payload	1-4	float	X component of Reference Gaze Vector
	5-8	float	Y component of Reference Gaze Vector
	9-12	float	Z component of Reference Gaze Vector
Response	0	uint8	0x85
	1	uint8	Return code

C

```
ah_result ah_api_runAutotune(ah_autotuneReferenceGazeVector *referenceGazeVec);
```

Python

```
api.trigger_autotune(reference_gaze_vector)
```

Re-center calibration [0x8f]

Updates an existing calibration based on the specified calibration point. The payload of the calibration data point indicates the position of the gaze target.

X, Y and Z is the position of the point relative to the midpoint of the scanners. See AdHawk Coordinate System for details.

API

	Byte	Type	Description
Request	0	uint8	0x8f
	1-4	float32	X position in meters
	5-8	float32	Y position in meters
	9-12	float32	Z position in meters
Response	0	uint8	0x8f
	1	uint8	Return code

C

```
ah_result ah_api_calibrationRecenter(float xPos, float yPos, float zPos);
```

Python

```
api.recenter_calibration(0.0, -0.179, -0.6)
```

Get tracker status [0x90]

Retrieves the current state of the eye-tracking module. A response value of zero (0) indicates that the tracker is calibrated and fully working.

API

	Byte	Type	Description
Request	0	uint8	0x90
Response	0	uint8	0x90
	1	uint8	Return code

C

```
ah_result ah_api_getTrackerStatus(void);
```

Python

```
try:
    api.get_tracker_status()
except adhawkapi.APIRequestError as exc:
    print(exc)
```

Note that this API returns the current tracker status, and will raise an exception in synchronous mode if the trackers are not ready.

We recommend registering for the tracker ready stream, which will notify any changes in the tracker status.

Start camera [0xd2]

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to turn on the front-facing camera.

API

	Byte	Type	Description
Request	0	uint8	0xd2
	1-2	uint16	Camera device index
	3-4	uint16	Image resolution index
	5	uint8	Undistort image (only recommended when a wide angle lens is used on the camera)
Response	0	uint8	0xd2
	1	uint8	Return code

Python

```
api.start_camera_capture(0, adhawkapi.CameraResolution.MEDIUM)
```

Stop camera [0xd3]

Currently not supported in c sdk

A stop command packet that will instruct AdHawk's eye-tracking module to turn off the front-facing camera.

API

	Byte	Type	Description
Request	0	uint8	0xd3
Response	0	uint8	0xd3
	1	uint8	Return code

Python

```
api.stop_camera_capture()
```

Subscribe for video stream [0xd4]

Currently not supported in c sdk

A command to subscribe to the video stream from the AdHawk front-facing camera. The receiver of the video needs to provide its IPv4 address and the port. See the example code in section Gaze In Image for how to setup a video receiver. Multiple receivers can subscribe to the video stream.

API

	Byte	Type	Description
Request	0	uint8	0xd4
	1-4	bytes	IPv4 Address of the receiver
	5-6	uint16	Port of the receiver
Response	0	uint8	0xd4
	1	uint8	Return code

Python

```
api.start_video_stream("192.168.0.1", 62829)
```

Unsubscribe from video stream [0xd5]

Currently not supported in c sdk

A command to unsubscribe for the video steam of the front facing camera given the ip and the port information of an existing video receiver. See the example code in section Gaze In Image for how to setup a video receiver.

API

	Byte	Type	Description
Request	0	uint8	0xd5
	1-4	bytes	IPv4 Address of the receiver
	5-6	uint16	port Port of the receiver
Response	0	uint8	0xd5
	1	uint8	Return code

Python

```
api.stop_video_stream("192.168.0.1", 62829)
```

Register Screen Aruco Board [0xda]

Currently not supported in c sdk

A packet will instruct AdHawk's eye-tracking module to reset any existing screen board defined for screen tracking and defines a new screen aruco board based on the given information. The packet includes the screen dimension (in meters), index of the aruco dictionary, and a set of markers (one or more) each defined by 3 float values indicating the bottom-left corner of the marker inside the board plane and the width of the marker. Multiple markers should be defined in the same packet. The coordinate system that is used to define the screen is located at the top-left corner of the screen with positive x axis toward the right and positive y axis upward. The bottom-left corner of each of the markers should be measured relative to that origin (in meters). Note that all the aruco markers should belong to the same aruco dictionary.

API

	Byte	Type	Description
Request	0	uint8	0xda
	1-4	float32	Width of the physical screen in meters
	5-8	float32	Height of the physical screen in meters
	9-10	uint16	Index of a OpenCV aruco predefined dictionary
	11-12	uint16	id of the aruco marker in the given dictionary
	13-16	float32	X position of bottom-left corner of the marker in meters relative to the top-left corner of the screen
	17-20	float32	Y position of bottom-left corner of the marker in meters relative to the top-left corner of the screen
	21-24	float32	size (width) of the marker in meters
			… repeat 11-24 for as many markers as there are being registered
Response	0	uint8	0xda
	1	uint8	Return code

Python

```
# registering a screen defined by a 20 cm marker located at 0, 0
api.register_screen_board(cv2.aruco.DICT_5X5_50, [1], [[0,0, 0.02]])
```

Screen tracking start [0xdb]

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to enable screen tracking once a screen has been registered.

API

	Byte	Type	Description
Request	0	uint8	0xdb
Response	0	uint8	0xdb
	1	uint8	Return code

Python

```
api.start_screen_tracking()
```

Screen tracking stop [0xdc]

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to stop screen tracking.

API

	Byte	Type	Description
Request	0	uint8	0xdc
Response	0	uint8	0xdc
	1	uint8	Return code

Python

```
api.stop_screen_tracking()
```

Run-time configuration objects (blobs)

The C SDK simplifies the use of blobs. This can be read / written to with the following APIs:

C

```
ah_result ah_api_readBlob(ah_blobType blobType, uint8_t **blobData, uint16_t *blobSize);

ah_result ah_api_writeBlob(ah_blobType blobType, uint8_t *blobData, uint16_t blobSize);
```

These functions use the appropriate commands detailed below.

Blob size [0x92]

Retrieves a run-time configuration object from the eye tracking module.

The client/host should get the size of the configuration object using get blob size command, and allocate the required buffer size to store the configuration content. Then, the client/host can get the blob data in chunks of max 25 bytes using the get blob data command until the entire configuration object is retrieved.

Get blob size

API

	Byte	Type	Description
Request	0	uint8	0x92
	1	uint8	Blob type
Response	0	uint8	0x92
	1	uint8	Return code
	2-3	uint16	The size of the blob

Python

```
errcode, blob_size = api.get_blob_size()
```

Set blob size

API

	Byte	Type	Description
Request	0	uint8	0x92
	1	uint8	Blob type

	Byte	Type	Description
	2-3	uint16	The size of the blob being written
Response	0	uint8	0x92
	1	uint8	Return code

Python

```
api.set_blob_size(adhawkapi.BlobType.CALIBRATION, len(blob_data))
```

Blob data [0x93]

Applies a run-time configuration object to the eye tracking module.

The client/host should set the size of the configuration object using the set blob size command. This will cause the eye tracking module to allocate the required buffer size to store the configuration content.Then the client/host can set the blob data in chunks of max 25 bytes using the set blob data command until the entire configuration object is written.

Get blob data

API

	Byte	Type	Description
Request	0	uint8	0x93
	1	uint8	Blob type
	2-3	uint16	The byte offset
Response	0	uint8	0x93
	1	uint8	Return code
	2..		Blob data (max 25 bytes)

Python

```
blob_type = adhawkapi.BlobType.CALIBRATION
_, size = api.get_blob_size(blob_type)
data = [None] * size
offset = 0

while offset <= size:
    _, chunk = api.get_blob_data(blob_type, offset)
    chunk_size = len(chunk)
    data[offset:offset + chunk_size] = chunk
    offset = offset + chunk_size
print(bytes(data))
```

Set blob data

API

	Byte	Type	Description
Request	0	uint8	0x93
	1	uint8	Blob type
	2-3	uint16	The byte offset
	4..		Blob data (max 25 bytes)

	Byte	Type	Description
Response	0	uint8	0x93
	1	uint8	Return code

Python

```
def chunks(data):
    '''Generator that breaks the data into chunks'''
    size = len(data)
    offset = 0
    chunk_len = adhawkapi.defaults.BLOB_CHUNK_LEN
    while (offset + chunk_len) <= size:
        yield offset, data[offset:offset + chunk_len]
        offset = offset + chunk_len
    yield offset, data[offset:]


blob_type = adhawkapi.BlobType.CALIBRATION
for offset, chunk in chunks(data):
    api.set_blob_data(blob_type, offset, chunk)
```

Load blob [0x94]

Applies a run-time configuration object to the eye tracking module. This is similar to writing a blob, except the data for the blob was previously saved to disk in the AdHawk Backend folder under cache as blob_<id>.bin

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0x94
	1	uint8	Blob type
	2-5	uint32	ID of the blob to load
Response	0	uint8	0x94
	1	uint8	Return code

Python

```
# Loading a calibration blob that was previously saved under cache/blob_1241123630.bin
api.load_blob(adhawkapi.BlobType.CALIBRATION, 1241123630)
```

Save blob [0x95]

Retrieves and saves a run-time configuration object from the eye tracking module. This is similar to reading a blob, except that the blob data is stored within the eye tracking module to disk in the AdHawk Backend folder under cache as blob_.bin, and the 32-bit unique identifier is returned to the clients for later use.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0x95
	1	uint8	Blob type
Response	0	uint8	0x95
	1	uint8	Return code
	2-5	uint32	ID of the saved blob

Python

```
errcode, blob_id = api.save_blob(adhawkapi.BlobType.CALIBRATION)
```

Properties

Get autotune position [0x9a, 0x01]

Get the center of the scan range. The returned values will be non-zero only if a trigger autotune command completed successfully or if the values were manually set through set autotune position

In the case of monocular operation, `0xFFFF` will be returned for the X mean and Y mean of the inactive eye

API

	Byte	Type	Description
Request	0	uint8	0x9a
	1	uint8	0x01
Response	0	uint8	0x9a
	1	uint8	Return code
	2	uint8	0x01
	3-4	uint16	X mean (right eye), valid range is 0 - 100
	5-6	uint16	Y mean (right eye), valid range is 0 - 100
	7-8	uint16	X mean (left eye), valid range is 0 - 100
	9-10	uint16	Y mean (left eye), valid range is 0 - 100

C

```
ah_result ah_api_getAutotunePosition(ah_autotunePosition pos[AH_NUM_EYES]);
```

Python

```
right_xmean, right_ymean, left_xmean, left_ymean = api.get_autotune_position()[2:]
```

Set autotune position [0x9b, 0x01]

Set the center of the scan range.

API

	Byte	Type	Description
Request	0	uint8	0x9b
	1	uint8	0x01
	2-3	uint16	X mean (right eye), valid range is 0 - 100
	4-5	uint16	Y mean (right eye), valid range is 0 - 100
	6-7	uint16	X mean (left eye), valid range is 0 - 100
	7-8	uint16	Y mean (left eye), valid range is 0 - 100

	Byte	Type	Description
Response	0	uint8	0x9b
	1	uint8	Return code
	2	uint8	0x01

C

```
ah_result ah_api_setAutotunePosition(const ah_autotunePosition pos[AH_NUM_EYES]);
```

Python

```
api.set_autotune_position(right_xmean, right_ymean, left_xmean, left_ymean)
```

Set stream control [0x9b, 0x02]

Control the rate of a data stream. Setting the rate to 0 disables the stream.

API

	Byte	Type	Description
Request	0	uint8	0x9b
	1	uint8	0x02
	2-5	uint32	Stream control bitmask
	6-9	float32	Stream rate in Hz. A value of 0 will disable the stream. See Supported rates
Response	0	uint8	0x9b
	1	uint8	Return code
	2	uint8	0x02

Example: Sending `0x9b, 0x02, 1 << 3, 60` enables the gaze stream and sets its rate to 60Hz. Sending `0x9b, 0x02, 1 << 3, 0` disables the gaze stream.

C

```
ah_result ah_api_setupStream(ah_streamType type, ah_streamRate rate, ah_streamCallback dataCb);
```

Python

```
api.set_stream_control(adhawkapi.PacketType.GAZE, 60)
```

Get stream control [0x9a, 0x02]

Get the current rate of a data stream

API

	Byte	Type	Description
Request	0	uint8	0x9a
	1	uint8	0x02
	0-3	uint32	Stream control bitmask (Only a single stream is allowed)
Response	0	uint8	0x9a
	1	uint8	Return code
	2	uint8	0x02
	3-6	float	Stream rate in Hz

Example: Sending `0x9a, 0x02, 1 << 3` will return `0x9a, 0x0, 0x2, 60` indicating that the rate for the gaze stream is currently 60 Hz.

C

```
ah_result ah_api_getStreamRate(ah_streamType type, ah_streamRate *streamRate);
```

Python

```
stream_rate = api.get_stream_control(adhawkapi.PacketType.GAZE)[2]
```

Get component offsets [0x9a, 0x04]

Get the currently configured component offsets. Component offsets can be used if the scanners are not in their nominal position. The offset is the position of the scanner relative to its nominal position. See AdHawk Coordinate System for details.

API

	Byte	Type	Description
Request	0	uint8	0x9a
	1	uint8	0x04
Response	0	uint8	0x9a
	1	uint8	Return code
	2	uint8	0x04
	3-6	float	right eye component offset x in millimeters
	7-10	float	right eye component offset y in millimeters
	11-14	float	right eye component offset z in millimeters
	15-18	float	left eye component offset x in millimeters
	19-22	float	left eye component offset y in millimeters
	23-26	float	left eye component offset z in millimeters

Python

```
rx, ry, rz, lx, ly, lz = api.get_component_offsets()[2:]
```

Set component offsets [0x9b, 0x04]

Configure the component offsets. Component offsets can be used if the scanners are not in their nominal position. The offset is the position of the scanner relative to its nominal position. See AdHawk Coordinate System for details.

API

	Byte	Type	Description
Request	0	uint8	0x9a
	1	uint8	0x04
Response	0	uint8	0x9a
	1	uint8	Return code
	2	uint8	0x04
	3-6	float	right eye component offset x in millimeters
	7-10	float	right eye component offset y in millimeters
	11-14	float	right eye component offset z in millimeters
	15-18	float	left eye component offset x in millimeters
	19-22	float	left eye component offset y in millimeters
	23-26	float	left eye component offset z in millimeters

Python

```
api.set_component_offsets(rx, ry, rz, lx, ly, lz)
```

Set event control [0x9b, 0x05]

Enable or disable an event stream.

API

	Byte	Type	Description
Request	0	uint8	0x9b
	1	uint32	0x05
	2-5	uint32	Event control bitmask
	6	uint32	0x01 to enable, 0x00 to disable
Response	0	uint8	0x9b
	1	uint8	Return code
	2	uint8	0x05

Get event control [0x9a, 0x05]

Get the currently enabled event streams

API

	Byte	Type	Description
Request	0	uint8	0x9a
	1	uint32	0x05
Response	0	uint8	0x9a
	1	uint8	Return code
	2	uint8	0x02
	3-6	uint32	Event control bitmask

Get normalized eye offsets [0x9a, 0x09]

Get the latest estimate of the eye offsets as measured from the nominal eye. The returned values will be valid only if a trigger autotune command completed successfully.

API

	Byte	Type	Description
Request	0	uint8	0x9a
	1	uint8	0x04
Response	0	uint8	0x9a
	1	uint8	Return code
	2	uint8	0x04
	3-6	float32	right eye x offset in millimeters
	7-10	float32	right eye y offset in millimeters
	11-14	float32	right eye z offset in millimeters
	15-18	float32	left eye x offset in millimeters
	19-22	float32	left eye y offset in millimeters
	23-26	float32	left eye z offset in millimeters

Python

```
rx, ry, rz, lx, ly, lz = api.get_normalized_eye_offsets()[2:]
```

System Control

Enable / Disable Eye Tracking [0x9c, 0x01]

Either enable or disable eye tracking

API

	Byte	Type	Description
Request	0	uint8	0x9c
	1	uint8	0x01
	2	uint8	0 to disable eye tracking, 1 to enable eye tracking
Response	0	uint8	0x9c
	1	uint8	Return code
	2	uint8	0x01

C

```
ah_result ah_api_enableTracking(bool enable);
```

Python

```
api.enable_tracking(True)
```

Device Setup Procedures

Trigger Device Calibration [0xb0, 0x01]

Begin a device calibration, which runs on all active trackers and restores tuning configurations to factory values. A response value of zero (0) indicates that the calibration has started successfully.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb0
	1	uint8	0x01 (device calibration procedure)
Response	0	uint8	0xb0
	1	uint8	Return code
	2	uint8	0x01

Python

```
api.trigger_device_calibration()
```

Get Device Calibration Status [0xb1, 0x01]

Retrieves the state of the current device calibration. If the calibration has not encountered an error, the returned value is an integer which indicates the progress of the current calibration, ranging from [0, 100], where 100 indicates the calibration has completed successfully.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb1
	1	uint8	0x01 (device calibration procedure)
Response	0	uint8	0xb1
	1	uint8	Return code
	2	uint8	0x01
	3	uint8	progress percentage

Python

```
errcode, _, status = api.get_device_calibration_status()
```

Trigger Update Firmware [0xb0, 0x02]

Begin a firmware update. A response value of zero (0) indicates that the update has started successfully.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb0
	1	uint8	0x02 (Update firmware procedure)
Response	0	uint8	0xb0
	1	uint8	Return code
	2	uint8	0x02

Python

```
api.trigger_update_firmware()
```

Get Update Firmware Status [0xb1, 0x02]

Retrieves the state of the current firmware update. If the update has not encountered an error, the returned value is an integer which indicates the progress of the update, ranging from [0, 100], where 100 indicates the update has completed successfully.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb1
	1	uint8	0x02 (Update firmware procedure)
Response	0	uint8	0xb1
	1	uint8	Return code
	2	uint8	0x02
	3	uint8	progress percentage

Python

```
errcode, _, status = api.get_update_firmware_status()
```

Marker Sequence Procedures

Calibration GUI Start [0xb0, 0x03]

Launch a calibration GUI provided by the AdHawk Backend Service to guide the user through a calibration procedure.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb0
	1	uint8	0x03 (Calibration procedure)
	2	uint8	Marker sequence mode
	3	uint8	Number of calibration points
	4-7	float32	Size of the marker in millimeter
	8	uint8	Randomize the sequence
	9-12	float32	Horizontal grid span in degrees of field of view (only for fixed-gaze modes)
	13-16	float32	Vertical grid span in degrees of field of view (only for fixed-gaze modes)
	17-20	float32	Horizontal shift of the grid in the user's field of view (only for fixed-gaze modes)
	21-24	float32	Vertical shift of the grid in the user's field of view (only for fixed-gaze modes)
Response	0	uint8	0xb0
	1	uint8	Return code
	2	uint8	0x03

Python

```
api.start_calibration_gui()
```

Get Calibration GUI Status [0xb1, 0x03]

Not supported

Validation GUI Start [0xb0, 0x04]

Launch a validation GUI provided by the AdHawk Backend Service to guide the user through a validation procedure.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb0
	1	uint8	0x04 (Validation procedure)
	2	uint8	Marker sequence mode
	3	uint8	Number of rows in the grid
	4	uint8	Number of columns in the grid
	5-8	float32	Size of the marker in millimeter
	9	uint8	Randomize the sequence
	10-13	float32	Horizontal grid span in degrees of field of view (only for fixed-gaze modes)

	Byte	Type	Description
	14-17	float32	Vertical grid span in degrees of field of view (only for fixed-gaze modes)
	18-21	float32	Horizontal shift of the grid in the user's field of view (only for fixed-gaze modes)
	22-25	float32	Vertical shift of the grid in the user's field of view (only for fixed-gaze modes)
Response	0	uint8	0xb0
	1	uint8	Return code
	2	uint8	0x04

Python

```
api.start_validation_gui()
```

Get Validation GUI Status [0xb1, 0x04]

Not supported

Autotune GUI Start [0xb0, 0x05]

Perform the autotune. A GUI is displayed to quickly tune the device.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb0
	1	uint8	0x05 (Autotune procedure)
	2	uint8	Marker sequence mode
	3-6	float32	Size of the marker in millimeter
Response	0	uint8	0xb0
	1	uint8	Return code
	2	uint8	0x05

Python

```
api.autotune_gui()
```

Get Autotune GUI Status [0xb1, 0x05]

Not supported

Quick-start GUI Start [0xb0, 0x06]

Perform the quick-start procedure. A GUI is displayed to quickly tune and calibrate the device.

Only supported when using the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xb0
	1	uint8	0x06 (Quick-start procedure)

	Byte	Type	Description
	2	uint8	Marker sequence mode
	3-6	float32	Size of the marker in millimeter
	7	uint8	returning user
Response	0	uint8	0xb0
	1	uint8	Return code
	2	uint8	0x06

Python

```
api.quick_start_gui()
```

Get Quick-start GUI Status [0xb1, 0x06]

Not supported

Camera user settings

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to set various camera-related settings used when calculating the Gaze-in-image or used when running the calibration/validation GUI provided by the AdHawk Backend Service that relies on the front-facing camera.

Set gaze depth [0xd0, 0x01]

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to set the default gaze depth used when projecting the user's gaze into the camera image. If not provided, the system uses a default value of 60 cm.

API

	Byte	Type	Description
Request	0	uint8	0xd0
	1	uint8	0x01 Camera user settings
	2-5	float32	gaze depth in meters
Response	0	uint8	0xd0
	1	uint8	Return code

Python

```
api.set_camera_user_settings(adhawkapi.CameraUserSettings.GAZE_DEPTH, 0.5)
```

Enable/Disable parallax correction [0xd0, 0x02]

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to enable or disable the parallax correction when projecting the user's gaze into the camera image. If not provided, the system enables the parallax correction by default.

API

	Byte	Type	Description
Request	0	uint8	0xd0
	1	uint8	0x02 Camera user settings
	2	uint8	enable (1) or disable (0) the parallax correction
Response	0	uint8	0xd0
	1	uint8	Return code

Python

```
api.set_camera_user_settings(adhawkapi.CameraUserSettings.PARALLAX_CORRECTION, 1)
```

Set sampling duration [0xd0, 0x03]

Currently not supported in c sdk

A command packet that will instruct AdHawk's eye-tracking module to set the sampling duration used during calibration or validation done by the GUI provided by the AdHawk Backend Service. If not provided, the system uses a default value of 500 ms.

API

	Byte	Type	Description
Request	0	uint8	0xd0
	1	uint8	0x03
	2-3	uint16	Sampling duration in milliseconds Camera user settings
Response	0	uint8	0xd0
	1	uint8	Return code

Python

```
api.set_camera_user_settings(adhawkapi.CameraUserSettings.SAMPLING_DURATION, 1000)
```

Backend service communication

Register endpoint [0xc0]

Register as an endpoint to start communication with the AdHawk Backend Service.

API

	Byte	Type	Description
Request	0	uint8	0xc0
	1	uint32	The UDP port of the peer
Response	0	uint8	0xc0
	1	uint8	Return code

Python

The packet construction is handled within the `start()` function of the frontendApi()

```
def on_connect(error):
    if not error:
        print('Connected to AdHawk Backend Service')
```

```
def on_disconnect(error):
    print('Disconnected from AdHawk Backend Service')

api.start(connect_cb=on_connect, disconnect_cb=on_disconnect)
```

Deregister endpoint [0xc2]

Deregister as an endpoint to stop communication with the AdHawk Backend Service

API

	Byte	Type	Description
Request	0	uint8	0xc2
Response	0	uint8	0xc2
	1	uint8	Return code

Python

```
api.shutdown()
```

Ping endpoint [0xc5]

Must be used to ping the AdHawk backend service to indicate that the frontend is still alive, and to verify that the backend service is still reachable. Should be sent every 2s. This is automatically handled within the Python and C SDKs.

API

	Byte	Type	Description
Request	0	uint8	0xc5
Response	0	uint8	0xc5
	1	uint8	Return code

Python

The `disconnect_cb` passed into start is executed if the pings aren't successful

Data Streams

Tracker ready [0x02]

The tracker ready packet type is an asynchronous status packet that is transmitted by AdHawk's eye-tracking module when the system is ready to receive calibration commands. Once the tracker ready signal packet is received, the client/host system may go through a calibration process (send Calibration start, followed by Register calibration data point, and finally Calibration complete).

	Byte	Type	Description
Stream	0	uint8	0x02

Python

```
tracker_ready_signal = threading.Event()
api.register_stream_handler(adhawkapi.PacketType.TRACKER_READY, lambda *args: tracker_ready_signal.set()
tracker_ready_signal.wait()
```

Gaze vector stream [0x03]

X, Y and Z are the estimated coordinates of the user's gaze point relative to the midpoint of the scanners. Vergence is the angle between left and right eye gaze vectors. See AdHawk Coordinate System for details.

API

	Byte	Type	Description
Stream	0	uint8	0x03
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	X position in meters
	9-12	float32	Y position in meters
	13-16	float32	Z position in meters
	17-20	float32	Vergence Angle in radians

C

```
typedef struct
{
    float xPos;
    float yPos;
    float zPos;
} ah_gazeData;

typedef struct __attribute__((__packed__))
{
    float timestamp;
    ah_gazeData gaze;
    float vergence;
} ah_gazeStreamData;
```

Python

```
def handler(*data):
    timestamp, xpos, ypos, zpos, vergence = data

api.register_stream_handler(adhawkapi.PacketType.GAZE, handler)
```

Pupil position stream [0x04]

X, Y and Z are coordinates of the pupil relative to the point between the center of the two (left and right) scanners. See AdHawk Coordinate System for details.

API

Binocular

	Byte	Type	Description
Stream	0	uint8	0x04
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	X position of right pupil in millimeters

	Byte	Type	Description
	9-12	float32	Y position of right pupil in millimeters
	13-16	float32	Z position of right pupil in millimeters
	17-20	float32	X position of left pupil in millimeters
	21-24	float32	Y position of left pupil in millimeters
	25-28	float32	Z position of left pupil in millimeters

Monocular

	Byte	Type	Description
Stream	0	uint8	0x04
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	X position of active pupil in millimeters
	9-12	float32	Y position of active pupil in millimeters
	13-16	float32	Z position of active pupil in millimeters

C

```c
typedef struct
{
    float x;
    float y;
    float z;
} ah_pupilPositionData;

typedef struct __attribute__((__packed__))
{
    float timestamp;
    ah_pupilPositionData pos[AH_NUM_EYES];
} ah_pupilPositionStreamData;
```

Python

```python
# Binocular
def handler(*data):
    timestamp, rx, ry, rz, lx, ly, lz = data

api.register_stream_handler(adhawkapi.PacketType.PUPIL_POSITION, handler)

# Monocular
def handler(*data):
    timestamp, x, y, z = data

api.register_stream_handler(adhawkapi.PacketType.PUPIL_POSITION, handler)
```

Pupil diameter stream [0x05]

The stream indicating the diameter of the pupil in millimeters

API

Binocular

	Byte	Type	Description
Stream	0	uint8	0x04
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	Diameter of right pupil in millimeters
	9-12	float32	Diameter of left pupil in millimeters

Monocular

	Byte	Type	Description
Stream	0	uint8	0x04
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	Diameter of active pupil in millimeters

C

```
struct ah_pupilSizeStreamData
{
    float timestamp;
    float diameter[AH_NUM_EYES];
};
```

Python

```
# Binocular
def handler(*data):
    timestamp, right_pupil, left_pupil = data

api.register_stream_handler(adhawkapi.PacketType.PUPIL_DIAMETER, handler)
api.set_stream_control(adhawkapi.PacketType.PUPIL_DIAMETER, rate)
```

```
# Monocular
def handler(*data):
    timestamp, pupil_diameter = data

api.register_stream_handler(adhawkapi.PacketType.PUPIL_DIAMETER, handler)
api.set_stream_control(adhawkapi.PacketType.PUPIL_DIAMETER, rate)
```

Per-eye gaze vector stream [0x06]

X, Y and Z are the coordinates of a unit vector relative to the center of each eye. See AdHawk Coordinate System for details.

API

Binocular

	Byte	Type	Description
Stream	0	uint8	0x06
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	X component of right eye gaze unit vector
	9-12	float32	Y component of right eye gaze unit vector
	13-16	float32	Z component of right eye gaze unit vector
	17-20	float32	X component of left eye gaze unit vector

	Byte	Type	Description
	21-24	float32	Y component of left eye gaze unit vector
	25-28	float32	Z component of left eye gaze unit vector

Monocular

	Byte	Type	Description
Stream	0	uint8	0x06
	1-4	float32	Timestamp since system start in seconds
	5-8	float32	X component of active eye gaze unit vector
	9-12	float32	Y component of active eye gaze unit vector
	13-16	float32	Z component of active eye gaze unit vector

C

```c
typedef struct
{
    float xPos;
    float yPos;
    float zPos;
} ah_gazeData;

typedef struct __attribute__((__packed__))
{
    float timestamp;
    ah_gazeData gaze[AH_NUM_EYES];
} ah_gazeStreamData;
```

Python

```python
# Binocular
def handler(*data):
    timestamp, rx, ry, rz, lx, ly, lz = data

api.register_stream_handler(adhawkapi.PacketType.PER_EYE_GAZE, handler)

# Monocular
def handler(*data):
    timestamp, x, y, z = data

api.register_stream_handler(adhawkapi.PacketType.PER_EYE_GAZE, handler)
```

Gaze in image stream [0x07]

This stream provides the (x, y) coordinates of the gaze projection in the camera image (see Image Coordinate System). The degree to pixels conversion coefficients are also included in the stream which are two values that indicate the equivalent pixel size (in the image) of a 1 degree of visual angle at the current gaze position.

API

	Byte	Type	Description
Stream	0	uint8	0x07
	1-4	float32	Timestamp since system start
	5-8	float32	x coordinate in the image

	Byte	Type	Description
	9-12	float32	y coordinate in the image
	13-16	float32	x degree to pixels
	17-20	float32	y degree to pixels

C

```
typedef struct
{
    float x;
    float y;
    float xDegToPix;
    float yDegToPix;
} ah_gazeInImageData;

typedef struct __attribute__((__packed__))
{
    float timestamp;
    ah_gazeInImageData gaze;
} ah_gazeInImageStreamData;
```

Python

```
def handler(*data):
    timestamp, xpos, ypos, xdegtopix, ydegtopix = data
    # use degtopix values to draw a gaze marker of a size equivalent to k degrees of visual angle. For
    # marker_size_deg = 1
    # marker_width_pixel = marker_size_deg * xdegtopix
    # marker_height_pixel = marker_size_deg * ydegtopix


api.register_stream_handler(adhawkapi.PacketType.GAZE_IN_IMAGE, handler)
api.set_stream_control(adhawkapi.PacketType.GAZE_IN_IMAGE, 1)
```

Gaze in screen stream [0x08]

This stream provides normalized (x, y) coordinates of the gaze inside the screen that was defined by a set of aruco markers. x and y are float values (0-1) where (0,0) is the top-left corner and (1, 1) is the bottom-right corner of the screen. See Register Screen Aruco Board for more details.

API

	Byte	Type	Description
Stream	0	uint8	0x07
	1-4	float32	Timestamp since system start
	5-8	float32	normalized x coordinate in the screen (value 0-1)
	9-12	float32	normalized y coordinate in the screen (value 0-1)

C

```
typedef struct
{
    float x;
    float y;
```

```
} ah_gazeInScreenData;

typedef struct __attribute__((__packed__))
{
    float timestamp;
    ah_gazeInScreenData gaze;
} ah_gazeInScreenStreamData;
```

Python

```
def handler(*data):
    timestamp, xpos, ypos = data

api.register_stream_handler(adhawkapi.PacketType.GAZE_IN_SCREEN, handler)
```

IMU data stream [0x17]

API

	Byte	Type	Description
Stream	0	uint8	0x17
	1-4	float32	Timestamp since system start
	5-8	float32	Gyro x (mdps)
	9-12	float32	Gyro y (mdps)
	13-16	float32	Gyro z (mdps)
	17-20	float32	Accelerometer x (mg)
	21-24	float32	Accelerometer y (mg)
	25-28	float32	Accelerometer z (mg)

Python

```
def handler(*data):
    timestamp, gx, gy, gz, accel_x, accel_y, accel_z = data

api.register_stream_handler(adhawkapi.PacketType.IMU, handler)
api.set_stream_control(adhawkapi.PacketType.IMU, rate)
```

IMU Rotation data stream [0x19]

API

	Byte	Type	Description
Stream	0	uint8	0x19
	1-4	float32	Timestamp since system start
	5-8	float32	X component of the rotation vector
	9-12	float32	Y component of the rotation vector
	13-16	float32	Z component of the rotation vector

C

Currently only available in UDP mode with backend

```
typedef struct __attribute__((__packed__))
{
    float timestamp;
```

```
    float Xrvec;
    float Yrvec;
    float Zrvec;
} ah_imuRotationStreamData;
```

Python

```
def handler(*data):
    timestamp, rvec_x, rvec_y, rvec_z = data

api.register_stream_handler(adhawkapi.PacketType.IMU_ROTATION, handler)
api.set_stream_control(adhawkapi.PacketType.IMU_ROTATION, rate)
```

Events

Blink events

Consists of 3 different events: Combined-eye blink event Per-eye eye close event Per-eye eye open event

Blink [0x18, 0x1]

Combined-eye blink event when the user's blink is detected on both eyes. More specifically, the blink event indicates the time window where both left and right blink events overlap in time. This event is triggered as soon as any of the closed eyes is opened.

API

	Byte	Type	Description
Event	0	uint8	0x18
	1	uint8	0x1 (Blink)
	2-5	float32	Timestamp since system start
	6-9	float32	Duration in ms

Eye Closed [0x18, 0x2]

Eye close event indicating that the eye (specified in the eye index) is closed.

API

	Byte	Type	Description
Event	0	uint8	0x18
	1	uint8	0x2 (Eye Closed)
	2-5	float32	Timestamp since system start
	6	uint8	Eye index (right=0, left=1)

Eye Opened [0x18, 0x3]

Eye open event indicating that the eye (specified in the eye index) is opened.

API

	Byte	Type	Description
Event	0	uint8	0x18
	1	uint8	0x3 (Eye Opened)

	Byte	Type	Description
	2-5	float32	Timestamp since system start
	6	uint8	Eye index (right=0, left=1)

Trackloss Events

Event indicating a trackloss of an eye. Prolonged blinks (exceeding the defined maximum duration of the blink) may be detected as trackloss.

Trackloss Start [0x18, 0x4]

Indicating the beginning of a trackloss event.

API

	Byte	Type	Description
Event	0	uint8	0x18
	1	uint8	0x4 (Track lost)
	2-5	float32	Timestamp since system start
	6	uint8	Eye index (right=0, left=1)

Trackloss End [0x18, 0x5]

Indicating the end of a trackloss event triggered as soon as the eye features are detected.

API

	Byte	Type	Description
Event	0	uint8	0x18
	1	uint8	0x5 (Eye detected)
	2-5	float32	Timestamp since system start
	6	uint8	Eye index (right=0, left=1)

Types reference

Return codes

API

Response	Description
0	Request successful
1	Internal failure
2	Invalid argument
3	Tracker not ready
4	No eyes detected
5	Right eye not detected
6	Left eye not detected
7	Not calibrated
8	Not supported
11	Request Timeout
12	Unexpected Response

Response	Description
13	Hardware Fault
14	Camera Fault
15	System Busy
16	Communication Error
17	Device Calibration Required

C

```c
typedef enum
{
    ah_result_Success,
    ah_result_Failure,
    ah_result_InvalidArgument,
    ah_result_TrackerNotReady,
    ah_result_EyesNotFound,
    ah_result_RightEyeNotFound,
    ah_result_LeftEyeNotFound,
    ah_result_NotCalibrated,
    ah_result_NotSupported,
    ah_result_SessionAlreadyRunning,
    ah_result_NoCurrentSession,
    ah_result_RequestTimeout,
    ah_result_UnexpectedResponse,
    ah_result_HardwareFault,
    ah_result_CameraFault,
    ah_result_Busy,
    ah_result_CommunicationError,
    ah_resultCode_DeviceCalibrationRequired,
} ah_result;
```

Python

```python
class AckCodes(enum.IntEnum):
    '''List of acknowledgement payload values'''

    SUCCESS = 0
    FAILURE = 1
    INVALID_ARGUMENT = 2
    TRACKER_NOT_READY = 3
    EYES_NOT_FOUND = 4
    RIGHT_EYE_NOT_FOUND = 5
    LEFT_EYE_NOT_FOUND = 6
    NOT_CALIBRATED = 7
    NOT_SUPPORTED = 8
    SESSION_ALREADY_RUNNING = 9
    NO_CURRENT_SESSION = 10
    REQUEST_TIMEOUT = 11
    UNEXPECTED_RESPONSE = 12
    HARDWARE_FAULT = 13
    CAMERA_FAULT = 14
    BUSY = 15
    COMMUNICATION_ERROR = 16
    DEVICE_CALIBRATION_REQUIRED = 17
```

Stream Control Bitmask

API

Bit	Stream Type
1	Pupil position
2	Pupil diameter
3	Gaze
4	Per Eye Gaze
31	IMU

C

```
typedef enum
{
    ah_streamType_Gaze,
    ah_streamType_PupilPosition,

    ah_streamType_Count
} ah_streamType;
```

Event Control Bitmask

API

Bit	Stream Type
0	Blink
1	Eye open / close
2	Trackloss start / end
7	External Trigger

Blob Type

API

Blob Type	Description
1	Gaze calibration data

Python

```
class BlobType(enum.IntEnum):
    '''Enum representing set of blob types'''
    CALIBRATION = 1
```

Supported Rates

The current listed of supported stream rates are: 30, 60, 125, 200, 250, 333, 500

C

```
typedef enum
{
    ah_streamRate_Off,
```

```
    ah_streamRate_30Hz,
    ah_streamRate_60Hz,
    ah_streamRate_125Hz,
    ah_streamRate_200Hz,
    ah_streamRate_250Hz,
    ah_streamRate_333Hz,
    ah_streamRate_500Hz,

    ah_streamRate_Count,
} ah_streamRate;
```

Python

```
class StreamRates(enum.IntEnum):
    '''Enum representing the set of supported rates'''
    OFF = 0
    RATE_30 = 30
    RATE_60 = 60
    RATE_125 = 125
    RATE_200 = 200
    RATE_250 = 250
    RATE_333 = 333
    RATE_500 = 500
```

Camera Resolution

Resolution index	Description
0	Low
1	Medium
2	High

Procedure Types

API

Procedure Type	Description
1	Device calibration
2	Firmware update
3	Backend service built-in calibration procedure with GUI
4	Backend service built-in validation procedure with GUI
5	Backend service built-in autotune procedure with GUI
6	Backend service built-in quickstart procedure with GUI

Python

```
class ProcedureType(enum.IntEnum):
    ''' Enum representing set of procedure types '''

    DEVICE_CALIBRATION = 1
    UPDATE_FIRMWARE = 2
    CALIBRATION_GUI = 3
    VALIDATION_GUI = 4
    AUTOTUNE_GUI = 5
```

```
QUICKSTART_GUI = 6
```

Marker Sequence Mode

The marker sequence GUI used during the backend gaze calibration procedure Marker Sequence Procedures can be launched in 4 modes (fixed-head or fixed-gaze modes where each can be either single marker or 4-markers). All of these modes show a set of visual markers in the screen and a small target for the user to fixate on when taking samples.

Mode index	Name
0	Fixed-head
1	Fixed-gaze
2	Fixed-head with 4 markers
3	Fixed-gaze with 4 markers

Fixed-head: A sequence of markers with a fixation target at the center will be shown on the screen and the user needs to look at the center of the target (while keeping their head still) and register a sample by pressing one of *enter*, *return*, or *space* keys.

Fixed-gaze: A sequence of markers will be shown at the center of the screen as well as an outline shape that moves on the screen as the user rotates their head. The user needs to align the outline shape with the target at the center while fixating at the center of the target. The user can then register the calibration sample by pressing the *space* key.

4-marker modes: The sampling procedure in these modes are identical to their corresponding above-mentioned modes. The only difference is that 4 markers will be displayed at the 4 corners of the screen, and they are separated from the fixation target. Use the 4-marker modes if the user finds focusing on the center of the aruco marker distracting.

Camera User Setting Types

Various camera-related settings used when calculating the Gaze-in-image or when running the built-in Calibration GUI provided by the AdHawk Backend Service (which relies on the camera).

Setting index	Description
1	Gaze depth
2	Parallax correction
3	Sampling duration

Extended Usage Notes

Python SDK

Synchronous vs Aysnchronous operation

In the Python SDK, you can chose to make the API calls blocking or non-blocking. To perform a non-blocking operation, provide a callback function as a parameter to be executed when the operation is complete. This mode of operation is useful for GUI applications. If a callback function is not provided, the call blocks till a response is received and returns the response on success or raises an exception on failure.

```
# blocking operation
try:
    api.trigger_autotune()
except adhawkapi.APIRequestError as err:
    print(err.ackcode)

# non-blocking operation
def on_autotune_complete(errcode):
    if errcode == adhawkapi.AckCodes.SUCCESS:
        print('Autotune complete')
    else:
        print(f'Autotune failed: {adhawkapi.errormsg(response)}')

api.trigger_autotune(callback=on_autotune_complete)
```

AdHawk Coordinate System

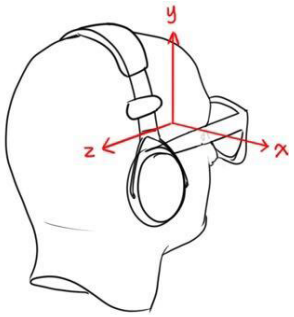

Figure 4: Coordinate System

In the AdHawk coordinate system X, Y and Z are coordinates relative to a particular origin where:

- X is oriented in the positive direction to the right (user's point of view)
- Y is oriented in the positive direction going up
- Z is oriented in the positive direction when behind the user

The origin is specific to the stream or command and is the same whether operating in monocular or binocular mode. In general the origin is either:

- The midpoint of the scanners
- The cyclopean eye
- The center of the eye

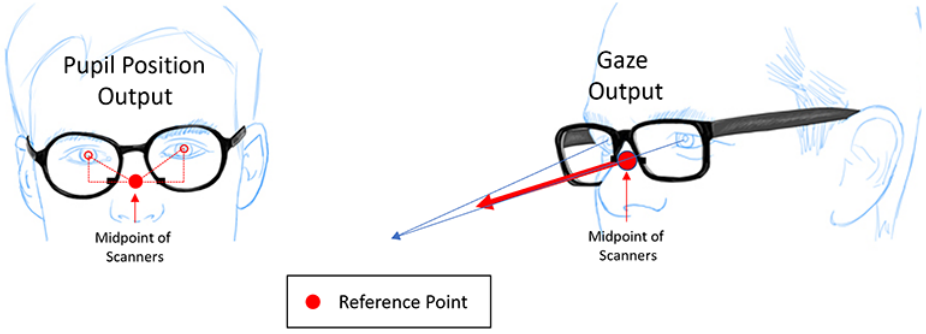

Figure 5: Origin

Stream	Origin
Gaze Vector	Midpoint of scanners
Pupil Position	Midpoint of scanners
Per-Eye Gaze Vector	Center of eye

Image Coordinate System

In the image coordinate system, X and Y are coordinates relative to the top-left corner of the image where:

- X is oriented in the positive direction to the right
- Y is oriented in the positive direction going down

Screen Coordinate System

In the image coordinate system, X and Y are coordinates relative to the top-left corner of the screen where:

- X is oriented in the positive direction to the right (user's point of view)
- Y is oriented in the positive direction going up

Changelog

v5.8

- Added Screen Tracking APIs:
  - Start Camera
  - Stop Camera
  - Subscribe to Video Stream
  - Unsubscribe to Video Stream
  - Register Screen Aruco Board
  - Screen tracking start
  - Screen tracking stop
  - Calibration GUI Start
  - Validation GUI Start
  - Autotune GUI Start
  - Quick-start GUI Start
  - Gaze in image stream
  - Gaze in screen stream
- Added Normalized Eye Offsets property

v5.9

- Added support for gaze reference vector input for autotune
  - Trigger Autotune