



## **pymkeapi.SyncClient v1.0 User Guide**

Magik Eye Inc.

COLLABORATORS			
	<i>TITLE :</i>  pymkeapi.SyncClient v1.0 User Guide		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Magik Eye Inc.	Jul 2020	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
2007-01-EN-00	Jul 2020	Initial version	MEI

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Setting Up the pymkeapi Wheel</b>	<b>1</b>
<b>3</b>	<b>Connecting to the Sensor</b>	<b>1</b>
<b>4</b>	<b>Making Mke API requests</b>	<b>1</b>
4.1	Mke API Constants . . . . .	2
4.2	Mke API Errors . . . . .	2
<b>5</b>	<b>Processing 3D Data</b>	<b>3</b>
<b>6</b>	<b>The get_frame () Method</b>	<b>3</b>
<b>7</b>	<b>The get_pushed_frame () Method</b>	<b>3</b>
<b>8</b>	<b>Reference</b>	<b>4</b>
<b>9</b>	<b>Bibliography</b>	<b>4</b>

## 1 Introduction

This document describes the `pymkeapi.SyncClient`, an implementation of a synchronous client of the MKE API v1.0 [\[mkeapi\]](#) in Python 3. This client is implemented by the `SyncClient` class that resides in the `pymkeapi` package.

The `SyncClient` class is a thin wrapper over the MKE API with an almost 1:1 correspondence between the API requests and the `SyncClient` methods. For this reason, this document does not explain the semantics of the API implementing methods and refers the reader to the *MagikEye API v1.0* documentation [\[mkeapi\]](#).

The officially supported system for the `pymkeapi` package is Ubuntu 20.04 64bit and all examples in this document execute on this operating system. However, the `pymkeapi` package itself should run on every system where Python 3 is available.

## 2 Setting Up the pymkeapi Wheel

The `pymkeapi` package is distributed as a wheel which can be readily installed---along with all its dependencies---via the `pip3` tool:

```
$ pip3 install pymkeapi-X.Y.Z-py3-none-any.whl
```

Here, `X.Y.Z` stands for the particular version of the package.

## 3 Connecting to the Sensor

A `SyncClient` object does not connect to a sensor directly. Rather, it uses another class from the `pymkeapi` package: `DefaultBus`. The `DefaultBus` class has two child classes, `TcpBus` and `SerialBus`, implementing TCP/IP and serial port style connections, respectively. The next code snippet shows how to connect to a sensor listening at `192.168.0.1:8888` through the TCP/IP network:

```
import pymkeapi
bus = pymkeapi.TcpBus(host='192.168.0.1', port=8888)
client = pymkeapi.SyncClient(bus)
```

Note that the `DefaultBus` can also be used as a context manager. The following code snippet shows how to use the `SerialBus` class to connect to an MKE API server via a serial port:

```
import pymkeapi
with pymkeapi.SerialBus(port='/dev/ttyS0') as bus:
    client = pymkeapi.SyncClient(bus)
```

## 4 Making MKE API requests

The MKE API requests are implemented via the `SyncClient` class methods, see [Section 8](#) for the full reference of the MKE API calls and the respective `SyncClient` methods.

The most basic example of an MKE API request is the `MKE_REQUEST_GET_STATE` request. This request is implemented by the `get_state()` method:

---

```
state = client.get_state()
```

The above code snippet expects a valid `SyncClient` object connected to an MKE API server. Note that since the `SyncClient` implements a synchronous MKE API client, it would not return immediately after sending the request data packet. Rather, it will wait for the reply data packet from the MKE server, check and parse the reply, and finally return the current state to user.

## 4.1 Mke API Constants

Every Mke API constant has its eponymous counterpart defined in the `pymkeapi` package. These can be easily explored using the `dir(pymkeapi)` function. The constants can be used instead of the numerical values in the python code. The next example uses the `pymkeapi.MKE_STATE_IDLE` constant to pass to the `MKE_STATE_IDLE` state:

```
state = client.get_state()
if state != pymkeapi.MKE_STATE_IDLE:
    client.set_state(pymkeapi.MKE_STATE_IDLE)
```

Another example is the sensor shutdown via the `MKE_REQUEST_TERMINATE` request:

```
client.terminate(pymkeapi.MKE_TERMINATE_BY_SHUTDOWN)
```

## 4.2 Mke API Errors

As mentioned in the above, once the `SyncClient` object makes an Mke API request, it waits for the server's reply. Upon receiving the reply, it checks the reply status. If the servers returned the `MKE_REPLY_OK` status, the `SyncClient` class continues to parse the reply data packet and returns the relevant data to the user. Otherwise, it raises an exception of type `pymkeapi.Error`. The `Error` object has an attribute `ret_code`, which is a numerical representation of the Mke API reply status code. The `ret_code_to_string()` method can be used to get the name of the reply status code as a string. Finally, one can simply convert an `Error` object to a string to get a sensible error message:

```
try:
    client.set_state(pymkeapi.MKE_STATE_IDLE)
except pymkeapi.Error as error:
    if error.ret_code == \
        pymkeapi.MKE_REPLY_CLIENT_REQUEST_DOES_NOT_APPLY:
        print('Cannot change sensor state: ' +
              error.ret_code_to_string(error.ret_code))
```

or, more simply:

```
try:
    client.set_state(pymkeapi.MKE_STATE_IDLE)
except pymkeapi.Error as error:
    print(error)
```

Note that the `SyncClient` class does not perform any validity checks of the parameters. Rather, it let's the Mke API server decide, if the parameter is valid or not. For example, the following code snippet will raise a `pymkeapi.Error` exception with the error code `MKE_REPLY_CLIENT_MALFORMED_REQUEST`, implying that the method's parameter is invalid:

```
try:
    client.set_state(100)
except pymkeapi.Error as error:
    if error.ret_code == pymkeapi.MKE_REPLY_CLIENT_MALFORMED_REQUEST:
        print('Invalid parameter: ' +
              error.ret_code_to_string(error.ret_code))
```

## 5 Processing 3D Data

The SyncClient supports both ways of receiving 3D data frames, *i.e.*,

- by *client polling* via the `get_frame()` method and
- by *sensor pushing* via the `get_pushed_frame()` method.

## 6 The `get_frame()` Method

Assuming the sensor is in the `MKE_STATE_DEPTH_SENSOR`, the `get_frame()` method will request and obtain one 3D data frame using the `MKE_REQUEST_GET_FRAME` request from the sensor:

```
frame = client.get_frame(pymkeapi.MKE_FRAME_TYPE_1)
```

The returned object `frame` is of the `pymkeapi.Frame` class. Among other things, the `pymkeapi.Frame` also contains properties `timer`, `seqn`, `data_type`, and `frame_type` whose values will correspond to the properties of `MkeReply_Frame`, see [\[mkeapi\]](#). Unlike the data payload of the reply to the `MKE_REQUEST_GET_FRAME` request, the `Frame` object does not contain a list of `MkeFrameItem1` or `MkeFrameItem2` structures. Rather, the structures are already parsed into numpy arrays, `uids` and `pts3d`, where the `x`, `y`, and `z` coordinates are stacked into a matrix.

## 7 The `get_pushed_frame()` Method

In order to initiate the sensor push of the 3D data frames, the `MKE_REQUEST_START_FRAME_PUSH` request must be made to the sensor. After this, the sensor will start pushing the 3D data frames without further client solicitation. The 3D frame data stream can be stopped using the `MKE_REQUEST_STOP_FRAME_PUSH` request. After this request, the client may still receive one or more replies with frame data, as some may have already been sent before the sensor received the `MKE_REQUEST_STOP_FRAME_PUSH` request, see [\[mkeapi\]](#) for more details. Using `SyncClient`, this translates into the following code snippet:

```
start_seq_id = client.start_frame_push(pymkeapi.MKE_FRAME_TYPE_1)
frame = client.get_pushed_frame(start_seq_id)

while stopping_condition != True:
    frame = client.get_pushed_frame(start_seq_id)
    # Process frame
```

```
stop_seq_id = client.stop_frame_push()

while frame is not None:
    frame = client.get_pushed_frame(start_seq_id, stop_seq_id)
```

The above code will initiate the sensor frame push and it will read the frames until a stopping\_condition is met. Note that the `get_pushed_frame()` method will block until a frame is received from the sensor. Also note that the `get_pushed_frame()` method is using the `start_seq_id` returned by `start_frame_push()` to connect the received frames to the correct stream. Finally, the stream will be stopped by the `stop_frame_push()` method. However, since some frames might have been already sent or are currently in the client's network buffer, the client needs to continue to receive them, now also using the `stop_seq_id` to correctly identify the end of the stream.

### Warning



It is important to keep the sequence of the `start_frame_push()`, `get_pushed_frame()`, and `stop_frame_push()` methods, together with the final `get_pushed_frame()` loop. This will ensure that the stream will be correctly stopped and all frames will be read out of the network buffer.

## 8 Reference

The following table contains the list of all available MKE API request and the respective `SyncClient` methods. See [\[mkeapi\]](#) for the semantics of the requests and their arguments.

<i>MkE API constant</i>	<i>SyncClient method</i>
MKE_REQUEST_TERMINATE	<code>terminate()</code>
MKE_REQUEST_GET_FIRMWARE_INFO	<code>get_fw_info()</code>
MKE_REQUEST_GET_DEVICE_INFO	<code>get_device_info()</code>
MKE_REQUEST_GET_DEVICE_XML	<code>get_device_xml()</code>
MKE_REQUEST_GET_STATE	<code>get_state()</code>
MKE_REQUEST_SET_STATE	<code>set_state()</code>
MKE_REQUEST_GET_POLICY	<code>get_policy()</code>
MKE_REQUEST_SET_POLICY	<code>set_policy()</code>
MKE_REQUEST_START_FRAME_PUSH	<code>start_frame_push()</code>
MKE_REQUEST_STOP_FRAME_PUSH	<code>stop_frame_push()</code>
MKE_REQUEST_GET_FRAME	<code>get_frame()</code>
MKE_REQUEST_LIST_POLICIES	<code>list_policies()</code>
MKE_REQUEST_UPLOAD_PACKAGE	<code>upload_package()</code>

## 9 Bibliography

- [\[ \] MagikEye API v1.0](#), 2020, Magik Eye Inc.