MkECLI User Guide

MKE API C++ Client v1.0

Magik Eye Inc.

COLLABORATORS

	TITLE : MkECLI User Guide	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Magik Eye Inc.	Mar 2021	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
2103-00-EN-00	Mar 2021		MEI

Contents

1 Introduction **1**

2 Building **1**

2.1 Building on Ubuntu 1

2.2 Linking on Ubuntu 1

2.3 Demo Applications 2

2.4 Building on Windows 2

2.5 Linking on Windows 2

3 Connecting to the Sensor **3**

4 MkE API Constants, Errors and Data Structures **3**

5 Making API requests **3**

5.1 Synchronous Interface Methods 4

5.2 Asynchronous Interface Methods 5

6 Processing 3D Data **5**

7 Framework Integration **6**

8 Bibliography **6**

1 Introduction

MkECLI, or `libmkeclient`, is a C++ implementation of the client-side MKE API v1.0 [mkeapi]. This document is `libmkeclient` user guide; it is not the library's reference manual. For a full reference, see the Doxygen generated source code documentation.

2 Building

The officially supported target platform is Ubuntu 20.04 running on an AMD64 system. However, the library can be compiled for other platforms as well, below is an example of compilation for Windows using MSVC compiler with CMake support.

The main dependency on `libmkeclient` is the `Boost`[1] library. The library also depends on the header-only `mkeapi` which provides definitions of all MkE API related structures though the `mkeapi.h` file. Both `mkeapi` and `libmkeclient` are part of the official MkE SDK and if compiled from the official SDK location, the `libmkeclient` build process is able to locate and use the `mkeapi` library automatically.

2.1 Building on Ubuntu

In order to compile the library on Ubuntu 20.04, the following packages are required:

```
$ sudo apt install build-essential git cmake libboost-all-dev
```

CMake is used as the build system for the library. Assuming that the `MKECLIENT_ROOT` environment variable points to the root directory of the `mkeclient` code base, to build the library, just execute the following commands in BASH:

```
$ mkdir ${MKECLIENT_ROOT}/build
$ cd ${MKECLIENT_ROOT}/build
$ cmake ..
$ make
```

The above commands should produce results in `build/src` folder.

Library and it's header files can then be installed to the system using the following command:

```
sudo make install
```

2.2 Linking on Ubuntu

Once the library is installed in the system, an application can be linked against it using the `-lmkeclient` linker flag. The following is an example of how to link code samples from this document, assuming that the `libmkeclient` library is installed using the steps from Section 2.1 and that the C++ source file is called `example.cpp`:

```
g++ -I/usr/local/include/mke/api/ -pthread example.cpp -o example \
    -lmkeclient -lboost_system -lboost_thread -lpthread
```

The above command should produce a binary executable called `example`.

[1] www.boost.org

2.3 Demo Applications

Several demo applications are built together with `libmkeclient` when the CMake option `MKECLI_DEMO` is selected. Also, the library verification tool can be built together with `libmkeclient` when the CMake option `MKECLI_TESTER` is selected. More about this testing tool can be found in its own documentation, see [tester].

```
$ mkdir ${MKECLIENT_ROOT}/build
$ cd ${MKECLIENT_ROOT}/build
$ cmake -DMKECLI_DEMO=ON -DMKECLI_TESTER=ON ..
$ make
```

2.4 Building on Windows

In order to build `libmkeclient` on Windows OS, download and install `Build Tools for Visual Studio 2019` or newer from `microsoft.com`.

Also, download and install the `Boost` library in version 1.74.0 or newer, see *e.g.*, from `sourceforge.net`.

In `x64 Native Tools Command Prompt for VS 2019`, navigate to the `libmkeclient` code base root folder (`mkeclient`) and execute the following commands:

```
mkdir build
cd build
cmake .. -A x64 -DMKECLI_TESTER=OFF -DMKECLI_DEMO=ON

msbuild ALL_BUILD.vcxproj /p:Configuration=Release
```

The above commands should produce results into the `build/src/Release` folder.

[optional] If you need to specify custom directories for `Boost` libraries, change the `cmake` command to:

```
cmake .. -A x64 -DMKECLI_TESTER=OFF -DMKECLI_DEMO=ON \
      -DBOOST_ROOT=C:\local\boost_1_74_0\boost \
      -DBOOST_INCLUDEDIR=C:\local\boost_1_74_0 \
      -DBOOST_LIBRARYDIR=C:\local\boost_1_74_0\lib64-msvc-14.2
```

[optional] If you want to create Debug builds, use:

```
msbuild ALL_BUILD.vcxproj /p:Configuration=Debug
```

2.5 Linking on Windows

Once the library is built using the steps from Section 2.4, an application can be linked against it using the following commands. Again, we are assuming that the source code is contained in the `example.cpp` file. In `x64 Native Tools Command Prompt for VS 2019`, navigate to the `libmkeclient` code base root folder (`mkeclient`) and execute the following commands:

```
SET MKECLIENT_SRC="src"
SET MKECLIENT_LIB="build/src/Release/mkeclient.lib"
SET MKEAPI_INCLUDE="../../../mkeapi/include"
cl /MT /EHsc /I%MKECLIENT_SRC% /I%MKEAPI_INCLUDE% example.cpp /link % ↩
   MKECLIENT_LIB% /MACHINE:X64
```

The above commands should produce the `example.exe` binary. Paths in above example assume default folder structure from the MkE SDK. If you want to build from elsewhere, the paths must be updated accordingly.

3 Connecting to the Sensor

The user of `libmkeclient` mainly communicates with the library through the class `mke::cli::Client`. However, the `Client` class does not implement the connection to the MkE API server [mkeapi] directly. Rather, it uses the `mke::cli::bus` class to manage the connection. The `bus` class has two child classes, `TcpBus` and `SerialBus`, implementing TCP/IP and serial port style connections, respectively. The following code snippet shows how to connect to a sensor listening at the IP address `192.168.0.1` and TCP port 8888:

```
#include "mke/cli/client.h"
#include "mke/cli/bus.h"

using namespace mke::cli;

int main(int argc, char* argv[]) {
  TcpBus tcp("192.168.0.1", 8888);
  Client client(&tcp);
  std::vector<char> buffer;
  client.setPayloadBuffer(&buffer);
  client.connect();
}
```

4 MkE API Constants, Errors and Data Structures

All the MkE API constants, errors, and data structures are defined in `mkeapi.h` which is a part of the `mkeapi` library.

Note

The `mke::cli::Client` class methods do not perform any validity checks of the parameters. Rather, the methods let the MkE API server decide if the parameter is valid or not.

---

5 Making API requests

There are two distinct ways of working with the `mke::cli::Client` class:

- by using the synchronous interface (methods with a timeout parameter) and
- by using the asynchronous interface (methods with callback parameters).

The actual communication is always asynchronous, hence the methods with the asynchronous interface provide a more direct access to the MkE API and can be faster in under some circumstances (mainly because the obtained data can be passed to the user without an additional copy). On the other

hand, the methods with the synchronous interface are simpler to use. Note that the synchronous interface methods are just convenience wrappers of their asynchronous equivalents. Both the synchronous and asynchronous methods can be used and combined interchangeably.

For the full reference of all available methods of the `mke::cli::Client` class see the doxygen generated documentation.

5.1 Synchronous Interface 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:

```
mke::api::MkEStateType state = client.getState();
```

The above code snippet expects a valid `client` object of the `mke::cli::Client` class connected to an MkE API server. Note that since the above call is perform using the synchronous version of the request, it will not return immediately after sending the request data packet. Rather, the method will wait for the reply data packet, check and parse the reply, and only after that return the current state to user.

All synchronous methods can be given a timeout parameter in milliseconds to limit the maximum blocking time for a response, *e.g.*:

```
mke::api::MkEStateType state = client.getState(500);
```

If a request fails for any reason (including reaching the given timeout), an exception will be thrown. The following is an example of error handling of a synchronous interface method:

```
try {
  mke::api::MkEStateType state = client.getState();
  std::cout << "The sensor state is: " << state << std::endl;
}
catch (mke::Error &e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

If needed, several exception types can be distinguished in order to customize the error handling for different situations, for example:

```
try {
  mke::api::MkEStateType state = client.getState();
  std::cout << "The sensor state is: " << state << std::endl;
}
catch (mke::cli::ServerFatalError &e) {
  std::cout << "Server fatal error: " << e.what() << std::endl;
}
catch (mke::cli::BadReplyError &e) {
  std::cout << "Bad reply error: " << e.what() << std::endl;
}
catch (mke::cli::IOError &e) {
  std::cout << "IO error: " << e.what() << std::endl;
}
catch (mke::Error &e) {
  std::cout << "Error: " << e.what() << std::endl;
}
```

Of course, multiple synchronous interface methods can share the same try-catch block. If synchronous and asynchronous interface methods are used together in the same code, please bear in mind that only synchronous interface methods throw exceptions. Asynchronous interface methods will call error callbacks in case of an error.

5.2 Asynchronous Interface Methods

The following example performs the exact same action as the one from Section 5.1. This time, the asynchronous interface method `getState()` is used:

```
mke::cli::StateCallback stateCallback =
 [](mke::api::MkEStateType state) {
  std::cout << "The sensor state is: " << state << std::endl;
};
mke::cli::ErrorCallback errorCallback =
 [](const mke::Error& error) {
  std::cout << "Error happened: " << error.what() << std::endl;
};
client.getState(stateCallback, errorCallback);
```

The above code can also be rewritten to use inline callbacks:

```
client.getState(
  [](mke::api::MkEStateType state) {
    std::cout << "The sensor state is: " << state << std::endl;
  },
  [](const mke::Error& error) {
    std::cout << "Error happened: " << error.what() << std::endl;
  });
```

The asynchronous interface methods return immediately after sending the request data packet. One of the provided callbacks will be called as soon as an MkE API reply is available or an error is encountered. Bear in mind that since the trivial example above doesn't contain any kind of a main loop, one would need to add at least a sleep call, *e.g*,

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

so the program doesn't end before the callbacks are called.

For more detailed examples on how to get the device state, set the device state or how to retrieve other information about the device, please refer to the library demo code `demo_device_state.cpp`.

6 Processing 3D Data

The `mke::cli::Client` class supports both MkE API methods [mkeapi] of receiving 3D data frames, *i.e*,

- by *client polling* via the `getFrame()` method and
- by *sensor pushing* via the `startFramePush()` method.

For examples on how to get and process frames via these methods, please refer to the library demo codes in the `${MKECLIENT_ROOT}/src/examples` subdirectory:

- demo_getframe_sync.cpp
- demo_getframe_async.cpp
- demo_pushframes.cpp.

In order to compile the example code in the ${MKECLIENT_ROOT}/src/examples subdirectory use the following BASH commands:

```
$ cd ${MKECLIENT_ROOT}/src/examples
$ mkdir build && cd build
$ cmake ..
$ make
```

The above example assumes that the libmkeclient library has been previously compiled within the ${MKECLIENT_ROOT} tree or has already been installed in the system.

7 Framework Integration

The libmkeclient library provides examples of integration of the data received from a sensor with several popular frameworks:

- Open3D, www.open3d.org
- OpenCV, www.opencv.org
- PCL, www.pointclouds.org

The example codes reside in the ${MKECLIENT_ROOT}/src/integration_examples subdirectory. A CMake compilation script is also provided: use MKE_USE_OPEN3D, MKE_USE_OPENCV, or MKE_USE_PCL CMake option to compile an example application for the respective platform. The MKE_ENABLE_VISU option enables visualization, in case such a tool is provided by the given framework, *e.g.*:

```
$ cd ${MKECLIENT_ROOT}/src/integration_examples
$ mkdir build && cd build
$ cmake -DMKE_USE_OPENCV=ON -DMKE_ENABLE_VISU=ON ..
$ make
```

The above example assumes that the libmkeclient library has been previously compiled within the ${MKECLIENT_ROOT} tree or has already been installed in the system.

8 Bibliography

- [] *MagikEye API v1.0*, 2020, Magik Eye Inc.
- [] *MkECLI Tester User Guide: MKE API C++ Client Tester v1.0*, 2021, Magik Eye Inc.