EMOTIV

GETTING STARTED GUIDE

Version 3.4.0

EMOTIV

Table of Contents

1. Getting started with Emotiv SDK

1.1 Overview

This section introduces key concepts for using the Emotiv SDK to build software that is compatible with Emotiv headsets. It also walks you through some sample programs that demonstrate these concepts and serve as a tutorial to help you get started with the Emotiv API. The sample programs are written in C++ and are intended to be compiled with Microsoft Visual Studio 2005 and can be found in the \doc\Examples directory of your installation or from our github page.

1.2 Introduction to the Emotiv API and Emotiv EmoEngine™

The Emotiv API is exposed as an ANSI C interface that is declared in 3 header files (iedk.h, iEmoStateDLL.h, iedkErrorCode.h) and implemented in a Window DLL named edk.dll. C or C++ applications that use the Emotiv API simply include iedk.h and link with edk.dll. See Appendix 4 for a complete description of redistributable Emotiv SDK components and installation requirements for your application.

The Emotiv EmoEngine refers to the logical abstraction of the functionality that Emotiv provides in edk.dll. The EmoEngine communicates with the Emotiv headset, receives preprocessed IEEG and gyroscope data, manages user-specific or application-specific settings, performs post-processing, and translates the Emotiv detection results into an easy-to-use structure called an EmoState. Emotiv API functions that modify or retrieve EmoEngine settings are prefixed with "IEE_."

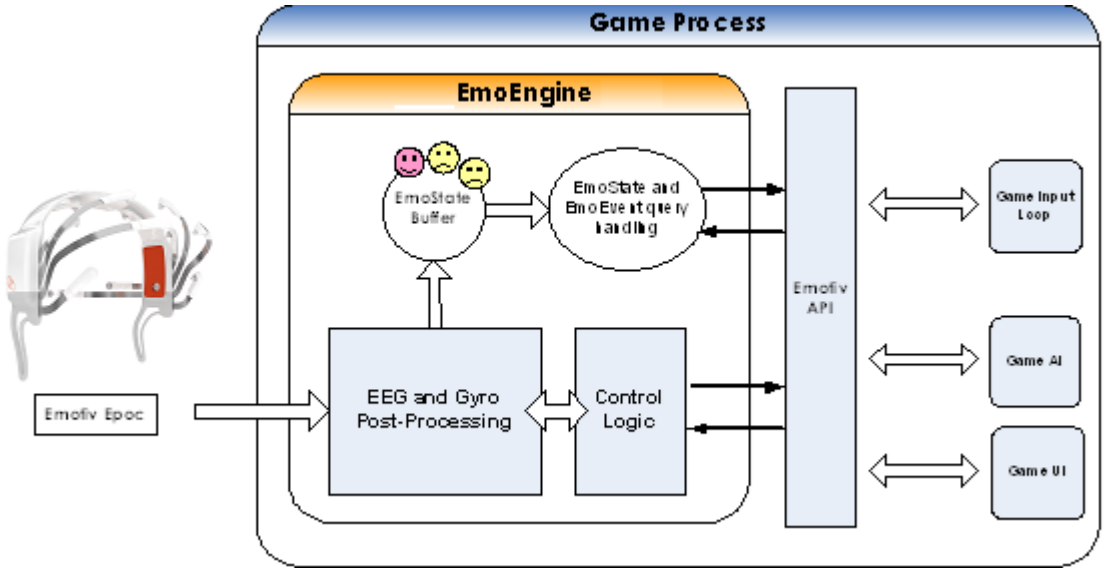


Figure 1 ***Integrating the EmoEngine and Emotiv EPOC with a videogame***

An EmoState is an opaque data structure that contains the current state of the Emotiv detections, which, in turn, reflect the user's facial, emotional and Mental Commands state. EmoState data is retrieved by Emotiv API functions that are prefixed with "IS_." EmoStates and other Emotiv API data structures are typically referenced through opaque handles (e.g. EmoStateHandle and EmoEngineEventHandle). These data structures and their handles are allocated and freed using the appropriate Emotiv API functions (e.g. IEE_EmoEngineEventCreate and IEE_EmoEngineEventFree).

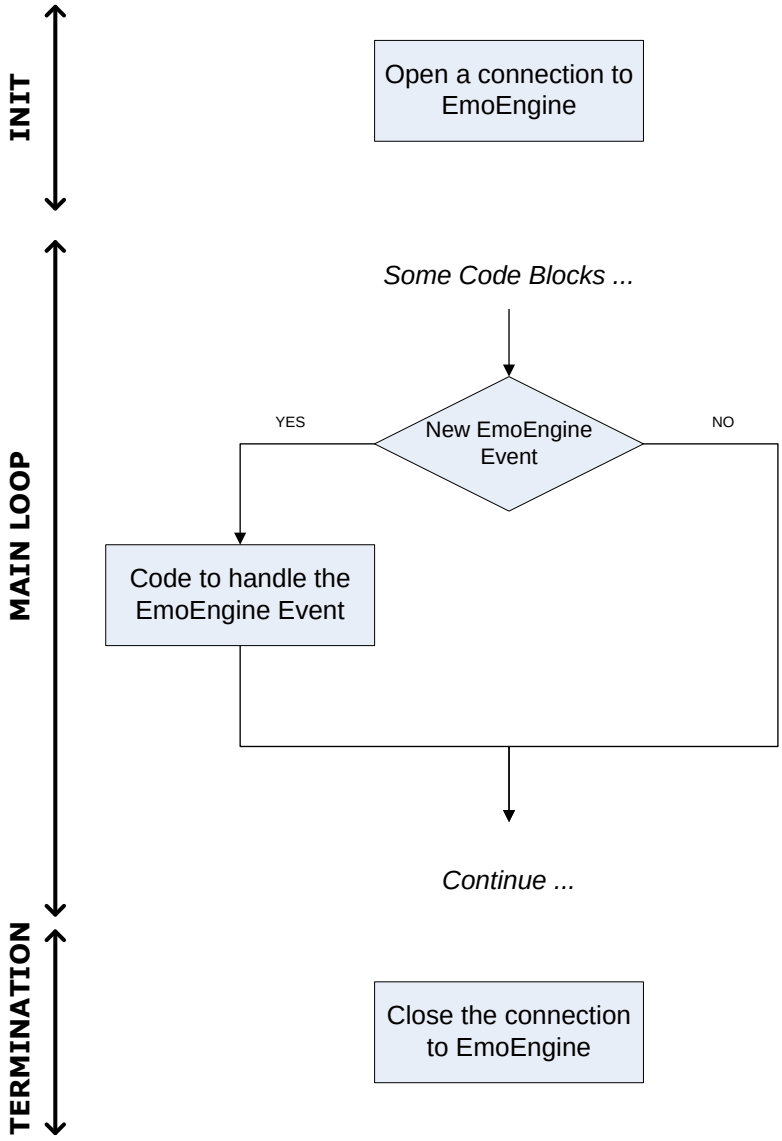


Figure 2 ***Using the API to communicate with the EmoEngine***

Figure 2 above shows a high-level flow chart for applications that incorporate the EmoEngine. During initialization, and prior to calling Emotiv API functions, your application must establish a connection to the EmoEngine by calling IEE_`EngineConnect` or IEE_`EngineRemoteConnect`. Use IEE_`EngineConnect` when you wish to communicate directly with an Emotiv headset. Use IEE_`EngineRemoteConnect` if you are using SDK and/or wish to connect your application to Composer or Emotiv. More details about using IEE_`EngineRemoteConnect` follow in Section 5.3.

The EmoEngine communicates with your application by publishing events that can be retrieved by calling IEE_EngineGetNextEvent(). For near real-time responsiveness, most applications should poll for new EmoStates at least 10-15 times per second. This is typically done in an application's main event loop or, in the case of most videogames, when other input devices are periodically queried. Before your application terminates, the connection to EmoEngine should be explicitly closed by calling IEE_EngineDisconnect().

There are three main categories of EmoEngine events that your application should handle:

- **Hardware-related events**: Events that communicate when users connect or disconnect Emotiv input devices to the computer (e.g. IEE_UserAdded).
- **New EmoState events**: Events that communicate changes in the user's facial, Mental Commands and emotional state. You can retrieve the updated EmoState by calling IEE_EmoEngineEventGetEmoState(). (e.g. IEE_EmoStateUpdated).
- **Suite-specific events**: Events related to training and configuring the Mental Commands and Facial Expressions detection suites (e.g. IEE_MentalCommandsEvent).

A complete list of all EmoEngine events can be found in **Error! Reference source not found.** .

Most Emotiv API functions are declared to return a value of type int. The return value should be checked to verify the correct operation of the API function call. Most Emotiv API functions return EDK_OK if they succeed. Error codes are defined in edkErrorCode.h and documented in **Error! Reference source not found.**.

1.3 Development Scenarios Supported by IEE_EngineRemoteConnect

The IEE_EngineRemoteConnect() API should be used in place of IEE_EngineConnect() in the following circumstances:

1. The application is being developed with Emotiv SDK. This version of the SDK does not include an Emotiv headset so all Emotiv API function calls communicate with Composer , the EmoEngine emulator that is described in Section **Error! Reference source not found.**. Composer listens on port 1726 so an application that wishes to connect to an instance of Composer running on the same computer must call. IEE_EngineRemoteConnect("127.0.0.1", 1726).
2. The developer wishes to test his application's behavior in a deterministic fashion by manually selecting which Emotiv detection results to send to the application. In this case, the developer should connect to Composer as described in the previous item.
3. The developer wants to speed the development process by beginning his application integration with the EmoEngine and the Emotiv headset without having to construct all of the UI and application logic required to support detection tuning, training, profile management and headset contact quality feedback. To support this case, Emotiv can act as a proxy for either the real, headset-integrated EmoEngine or Composer. SDK listens on port 3008 so an application that wishes to connect to SDK must call IEE_EngineRemoteConnect("127.0.0.1", 3008).
4. Emotiv SDK uses function:

   EDK_API int

   IEE_HardwareGetVersion(unsigned int userId, unsigned long* pHwVersionOut);

   This function will return the current hardware version of the headset and dongle for a particular user.

   \param pHwVersionOut - hardware version for the user headset/dongle pair. hiword is headset version, loword is dongle version.

   \return EDK_ERROR_CODE

   - EDK_ERROR_CODE = EDK_OK if successful

   \sa iEmoStateDll.h,i edkErrorCode.h

   (Latest firmware information can be found in API reference)

1.4 Example 1 – EmoStateLogger

This example demonstrates the use of the core Emotiv API functions described in Sections 1.2 and 3.3 It logs all Emotiv detection results for the attached users after successfully establishing a connection to Emotiv EmoEngine™ or Composer ™.

```
// ... print some instructions...
std::string input;
std::getline(std::cin, input, '\n');
option = atoi(input.c_str());

switch (option) {
    case 1: {
        if (IEE_EngineConnect() != EDK_OK) {
            throw exception("Emotiv Engine start up failed.");
        }
        break;
    }
    case 2: {
        std::cout << "Target IP of Composer ? [127.0.0.1] ";
        std::getline(std::cin, input, '\n');
        if (input.empty()) {
            input = std::string("127.0.0.1");
        }
        if (IEE_EngineRemoteConnect(input.c_str(), 1726) != EDK_OK){
            throw exception("Cannot connect to Composer !");
        }
        break;
    }
    default:
        throw exception("Invalid option...");
        break;
}
```

Listing 1 ***Connect to the EmoEngine***

The program first initializes the connection with Emotiv EmoEngine™ by calling `IEE_EngineConnect()` or, with InsightComposer, via `IEE_EngineRemoteConnect()` together with the target IP address of the Composer machine and the fixed port 1726. It ensures that the remote connection has been successfully established by verifying the return value of the `IEE_EngineRemoteConnect()` function.

```
EmoEngineEventHandle IEEvent   = IEE_EmoEngineEventCreate();
EmoStateHandle eState          = IEE_EmoStateCreate();
unsigned int userID            = 0;
while (...) {
int state = IEE_EngineGetNextEvent(eEvent);
// New event needs to be handled
if (state == EDK_OK) {
        IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
        IEE_EmoEngineEventGetUserId(eEvent, &userID);
        // Log the EmoState if it has been updated
        if (eventType == IEE_EmoStateUpdated) {
                // New EmoState from user
                IEE_EmoEngineEventGetEmoState(eEvent, eState);
                // Log the new EmoState
                logEmoState(ofs, userID, eState, writeHeader);
                writeHeader = false;
        }
    }
}
```

Listing 2 ***Buffer creation and management***

An `EmoEngineEventHandle` is created by `IEE_EmoEngineEventCreate()`. An EmoState™ buffer is created by calling `IEE_EmoStateCreate()`. The program then queries the EmoEngine to get the current EmoEngine event by invoking `IEE_EngineGetNextEvent()`. If the result of getting the event type using `IEE_EmoEngineEventGetType()` is `IEE_EmoStateUpdated`, then there is a new detection event for a particular user (extract via `IEE_EmoEngineEventGetUserID()`). The function `IEE_EmoEngineEventGetEmoState()` can be used to copy the EmoState™ information from the event handle into the pre-allocated EmoState buffer.

Note that IEE_EngineGetNextEvent() will return `EDK_NO_EVENT` if no new events have been published by EmoEngine since the previous call. The user should also check for other error codes returned from `IEE_EngineGetNextEvent()` to handle potential problems that are reported by the EmoEngine.

Specific detection results are retrieved from an EmoState by calling the corresponding EmoState accessor functions defined in `EmoState.h`. For example, to access the blink detection, `IS_FacialExpressivIsBlink(eState)` should be used.

```
IEE_EngineDisconnect();
IEE_EmoStateFree(eState);
IEE_EmoEngineEventFree(eEvent);
```

Listing 3 ***Disconnecting from the EmoEngine***

Before the end of the program, `IEE_EngineDisconnect()` is called to terminate the connection with the EmoEngine and free up resources associated with the connection. The user should also call `IEE_EmoStateFree()` and `IEE_EmoEngineEventFree()` to free up memory allocated for the EmoState buffer and `EmoEngineEventHandle`.

Before compiling the example, use the **Property Pages** and set the **Configuration Properties→Debugging→Command Arguments** to the name of the log file you wish to create, such as **log.txt**, and then build the example.

To test the example, launch Composer. Start a new instance of EmoStateLogger and when prompted, select option 2 (***Connect to Composer)***. The EmoStates generated by Composer will then be logged to the file **log.txt**.

Tip: If you examine the log file, and it is empty, it may be because you have not used the controls in the Composer to generate any EmoStates. SDK users should only choose option 2 to connect to Composer since option 1 (***Connect to EmoEngine***) assumes that the user will attach a neuroheadset to the computer.

1.5 Example 2 – Facial Expressions Demo

This example demonstrates how an application can use the Facial Expressions detection suite to control an animated head model called BlueAvatar. The model emulates the facial expressions made by the user wearing an Emotiv headset. As in Example 1, Facial Expressions Demo connects to Emotiv EmoEngine™ and retrieves EmoStates™ for all attached users. The EmoState is examined to determine which facial expression best matches the user's face. Facial Expressions Demo communicates the detected expressions to the separate BlueAvatar application by sending a UDP packet which follows a simple, pre-defined protocol.

The Facial Expressions state from the EmoEngine can be separated into three groups of mutually-exclusive facial expressions:

- **Upper face actions**: Surprise, Frown
- **Eye related actions**: Blink, Wink left, Wink right
- **Lower face actions**: Smile, Clench, Laugh

```
EmoStateHandle eState = IEE_EmoStateCreate();
...
IEE_FacialExpressivAlgo_t upperFaceType = IS_FacialExpressivGetUpperFaceAction(eState);
IEE_FacialExpressivAlgo_t lowerFaceType = IS_FacialExpressivGetLowerFaceAction(eState);
float upperFaceAmp = IS_FacialExpressivGetUpperFaceActionPower(eState);
float lowerFaceAmp = IS_FacialExpressivGetLowerFaceActionPower(eState);
```

Listing 4 ***Excerpt from Facial Expressions Demo code***

This code fragment from Facial Expressions Demo shows how upper and lower face actions can be extracted from an EmoState buffer using the Emotiv API functions `IS_FacialExpressivGetUpperFaceAction()` and `IS_FacialExpressivGetLowerFaceAction()`, respectively. In order to describe the upper and lower face actions more precisely, a floating point value ranging from 0.0 to 1.0 is associated with each action to express its "power", or degree of movement, and can be extracted via the `IS_FacialExpressivGetUpperFaceActionPower()` and `IS_FacialExpressivGetLowerFaceActionPower()` functions.

Eye and eyelid-related state can be accessed via the API functions which contain the corresponding expression name such as `IS_FacialExpressivIsBlink()`, `IS_FacialExpressivIsLeftWink()` etc.

The protocol that Facial Expressions Demo uses to control the BlueAvatar motion is very simple. Each facial expression result will be translated to plain ASCII text, with the letter prefix describing the type of expression, optionally followed by the amplitude value if it is an upper or lower face action. Multiple expressions can be sent to the head model at the same time in a comma separated form. However, only one expression per Facial Expressions grouping is permitted (the effects of sending smile and clench together or blinking while winking is undefined by the BlueAvatar). Table 1 below excerpts the syntax of some of expressions supported by the protocol.

Facial Expressions action type	Corresponding ASCII Text (case sensitive)	Amplitude value
Blink	B	n/a
Wink left	l	n/a
Wink right	r	n/a
Surprise	b	0 to 100 integer
Frown	F	0 to 100 integer
Smile	S	0 to 100 integer
Clench	G	0 to 100 integer

Table 1 ***BlueAvatar control syntax***

Some examples:

- Blink and smile with amplitude 0.5: **B,S50**
- Surprise and Frown with amplitude 0.6 and clench with amplitude 0.3: **b60, G30**
- Wink left and smile with amplitude 1.0: **l, S100**

The prepared ASCII text is subsequently sent to the BlueAvatar via UDP socket. Facial Expressions Demo supports sending expression strings for multiple users. BlueAvatar should start listening to port 30000 for the first user. Whenever a subsequent Emotiv USB receiver is plugged-in, Facial Expressions Demo will increment the target port number of the associated BlueAvatar application by one. Tip: when an Emotiv USB receiver is removed and then reinserted, Facial Expressions Demo will consider this as a new Emotiv EPOC and still increases the sending UDP port by one.

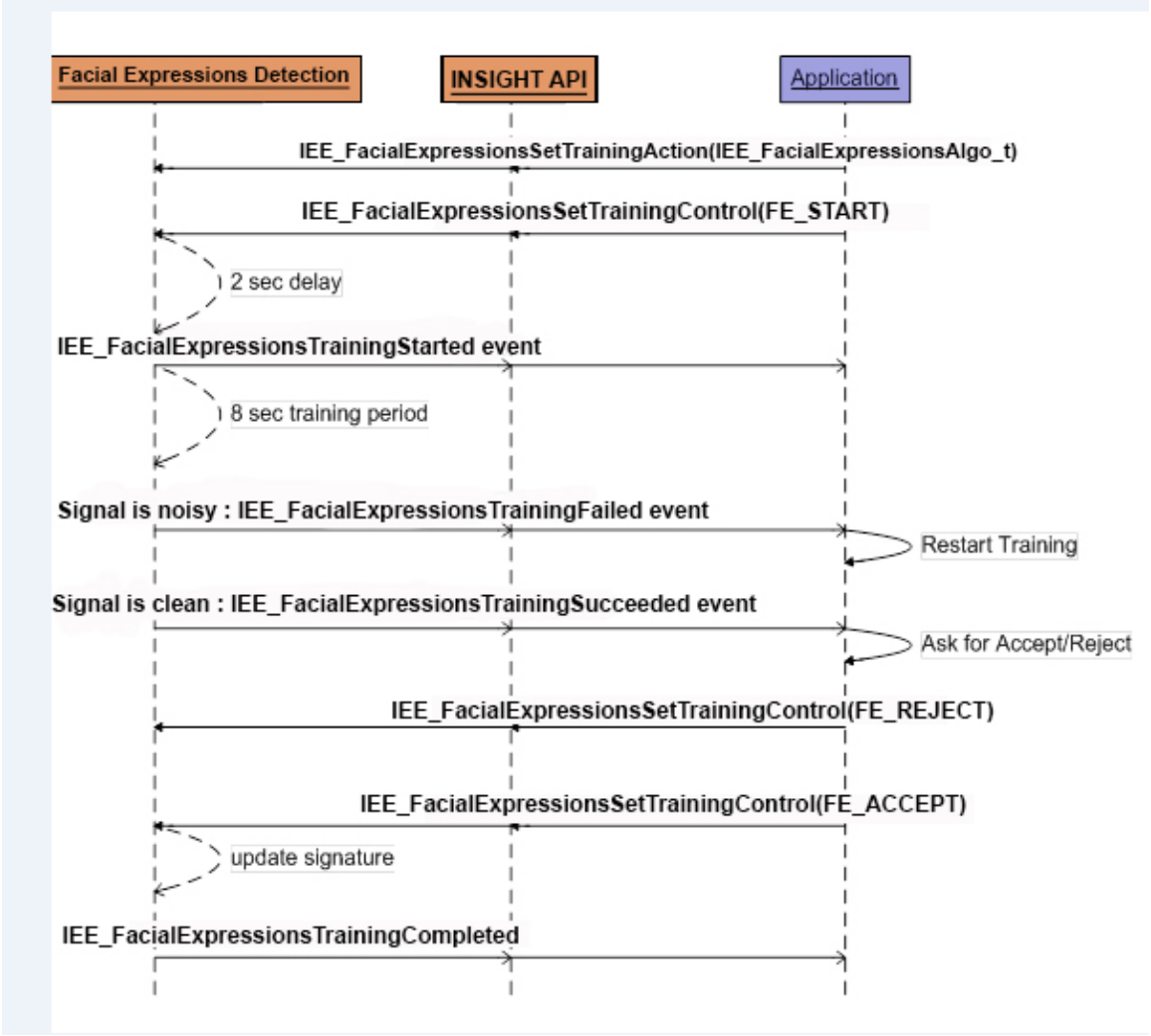
In addition to translating Facial Expressions results into commands to the BlueAvatar, the Facial Expressions Demo also implements a very simple command-line interpreter that can be used to demonstrate the use of personalized, trained signatures with the Facial Expressions. Facial Expressions supports two types of "signatures" that are used to classify input from the Emotiv headset as indicating a particular facial expression.

The default signature is known as the universal signature, and it is designed to work well for a large population of users for the supported facial expressions. If the application or user requires more accuracy or customization, then you may decide to use a trained signature. In this mode, Facial Expressions requires the user to train the system by performing the desired action before it can be detected. As the user supplies more training data, the accuracy of the Facial Expressions detection typically improves. If you elect to use a trained signature, the system will only detect actions for which the user has supplied training data. The user must provide training data for a neutral expression and at least one other supported expression before the trained signature can be activated. Important note: not all Facial Expressions expressions can be trained. In particular, eye and eyelid-related expressions (i.e. "blink", "wink") cannot be trained.

The API functions that configure the Facial Expressions detections are prefixed with "IEE_FacialExpressiv." The **training_exp** command corresponds to the `IEE_FacialExpressivSetTrainingAction() function`. The **trained_sig** command corresponds to the `IEE_FacialExpressivGetTrainedSignatureAvailable() function`. Type "help" at the Facial Expressions Demo command prompt to see a complete set of supported commands.

The figure below illustrates the function call and event sequence required to record training data for use with Facial Expressions. It will be useful to first familiarize yourself with the training procedure

on the Facial Expressions tab in Emotiv before attempting to use the Facial Expressions training API functions.



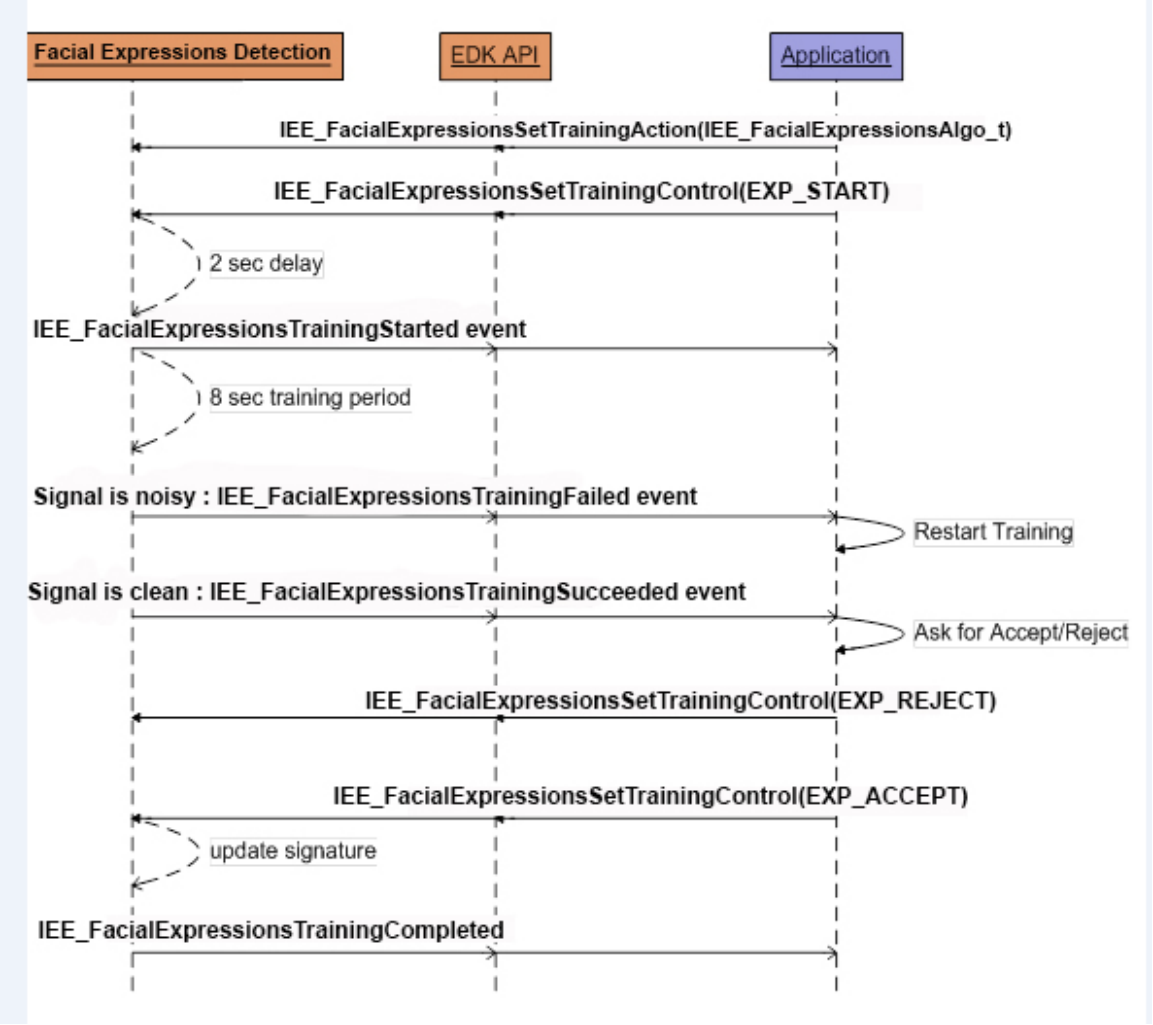


Figure 3 ***Facial Expressions training command and event sequence***

The below sequence diagram describes the process of training an Facial Expressions facial expression. The Facial Expressions -specific training events are declared as enumerated type `IEE_FacialExpressivEvent_t` in i`EDK.h`. Note that this type differs from the `IEE_Event_t` type used by top-level EmoEngine Events.

```
EmoEngineEventHandle IEEvent   = IEE_EmoEngineEventCreate();
if (IEE_EngineGetNextEvent(eEvent) == EDK_OK) {
    IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
    if (eventType == IEE_FacialExpressivEvent) {
            IEE_FacialExpressivEvent_t cEvt=IEE_FacialExpressivEventGetType(eEvent);
            ...
}
```

Listing 5 ***Extracting Facial Expressions event details***

Before the start of a training session, the expression type must be first set with the API function `IEE_FacialExpressivSetTrainingAction()`. In `iEmoStateDLL.h`, the enumerated type `IEE_FacialExpressivAlgo_t` defines all the expressions supported for detection. Please note, however, that only non-eye-related detections (lower face and upper face) can be trained. If an expression is not set before the start of training, `EXP_NEUTRAL` will be used as the default.

`IEE_FacialExpressivSetTrainingControl()` can then be called with argument `EXP_START` to start the training the target expression. In `iEDK.h`, enumerated type `IEE_FacialExpressivTrainingControl_t` defines the control command constants for Facial Expressions training. If the training can be started, an `IEE_FacialExpressivTrainingStarted` event will be sent after approximately 2 seconds. The user should be prompted to engage and hold the desired facial expression prior to sending the `EXP_START command`. The training update will begin after the EmoEngine sends the `IEE_FacialExpressivTrainingStarted` event. This delay will help to avoid training with undesirable IEEG artifacts resulting from transitioning from the user's current expression to the intended facial expression.

After approximately 8 seconds, two possible events will be sent from the EmoEngine™:

`IEE_FacialExpressivTrainingSucceeded`: If the quality of the IEEG signal during the training session was sufficiently good to update the Facial Expressions algorithm's trained signature, the EmoEngine will enter a waiting state to confirm the training update, which will be explained below.

`IEE_FacialExpressivTrainingFailed`: If the quality of the IEEG signal during the training session was not good enough to update the trained signature then the Facial Expressions training process will be reset automatically, and user should be asked to start the training again.

If the training session succeeded (`IEE_FacialExpressivTrainingSucceeded` was received) then the user should be asked whether to accept or reject the session. The user may wish to reject the training session if he feels that he was unable to maintain the desired expression throughout the duration of the training period. The user's response is then submitted to the EmoEngine through the API call `IEE_FacialExpressivSetTrainingControl()` with argument `FE_ACCEPT` or `FE_REJECT`. If the training is rejected, then the application should wait until it receives the `IEE_FacialExpressivTrainingRejected` event before restarting the training process. If the training is accepted, EmoEngine™ will rebuild the user's trained Facial Expressions signature, and an `IEE_FacialExpressivTrainingCompleted` event will be sent out once the calibration is done. Note that this signature building process may take up several seconds depending on system resources, the number of expression being trained, and the number of training sessions recorded for each expression.

To run the Facial Expressions Demo example, launch the Emotiv and Composer. In the Emotiv select **Connect→To Composer** , accept the default values and then enter a new profile name. Next, navigate to the doc\Examples\example2\blueavatar folder and launch the BlueAvatar application. Enter 30000 as the UDP port and press the **Start Listening** button. Finally, start a new

instance of Facial Expressions Demo, and observe that when you use the **Upperface**, **Lowerface** or **Eye** controls in Composer, the BlueAvatar model responds accordingly.

Next, experiment with the training commands available in Facial Expressions Demo to better understand the Facial Expressions training procedure described above. Listing 6 shows a sample Facial Expressions Demo sessions that demonstrates how to train an expression.

```
Emotiv Engine started!
Type "exit" to quit, "help" to list available commands...
FacialExpressionsDemo>
New user 0 added, sending Facial Expressions animation to localhost:30000...
FacialExpressionsDemo> trained_sig 0
==> Querying availability of a trained Facial Expressions signature for user 0...
A trained Facial Expressions signature is not available for user 0

FacialExpressionsDemo> training_exp 0 neutral
==> Setting Facial Expressions training expression for user 0 to neutral...

FacialExpressionsDemo> training_start 0
==> Start Facial Expressions training for user 0...

FacialExpressionsDemo>
Facial Expressions training for user 0 STARTED!
FacialExpressionsDemo>
Facial Expressions training for user 0 SUCCEEDED!
FacialExpressionsDemo> training_accept 0
==> Accepting Facial Expressions training for user 0...

FacialExpressionsDemo>
Facial Expressions training for user 0 COMPLETED!
FacialExpressionsDemo> training_exp 0 smile
==> Setting Facial Expressions training expression for user 0 to smile...

FacialExpressionsDemo> training_start 0
==> Start Facial Expressions training for user 0...

FacialExpressionsDemo>
Facial Expressions training for user 0 STARTED!
FacialExpressionsDemo>
Facial Expressions training for user 0 SUCCEEDED!
FacialExpressionsDemo> training_accept 0
==> Accepting Facial Expressions training for user 0...

FacialExpressionsDemo>
Facial Expressions training for user 0 COMPLETED!
FacialExpressionsDemo> trained_sig 0
==> Querying availability of a trained Facial Expressions signature for user 0...
A trained Facial Expressions signature is  available for user 0

FacialExpressionsDemo> set_sig 0 1
==> Switching to a trained Facial Expressions signature for user 0...
```

```
FacialExpressionsDemo>
```

Listing 6 ***Training "smile" and "neutral" in Facial Expressions Demo***

1.6 Example 3 – Profile Management

User-specific detection settings, including trained Mental Commands and Facial Expressions signature data, currently enabled Mental Commands actions, Mental Commands and Facial Expressions sensitivity settings, and Performance Metrics calibration data, are saved in a user profile that can be retrieved from the EmoEngine and restored at a later time.

This example demonstrates the API functions that can be used to manage a user's profile within Emotiv EmoEngine™. Please note that this example requires the Boost C++ Library in order to build correctly. Boost is a modern, open source, peer-reviewed, C++ library with many powerful and useful components for general-purpose, cross-platform development. For more information and detailed instructions on installing the Boost library please visit http://www.boost.org.

```
if (IEE_EngineConnect() == EDK_OK) {
   // Allocate an internal structure to hold profile data
   EmoEngineEventHandle eProfile          = IEE_ProfileEventCreate();
     // Retrieve the base profile and attach it to the eProfile handle
     IEE_GetBaseProfile(eProfile);
}
```

Listing 7 ***Retrieve the base profile***

`IEE_EngineConnect() or IEE_EngineRemoteConnect()` must be called before manipulating EmoEngine profiles. Profiles are attached to a special kind of event handle that is constructed by calling `IEE_ProfileEventCreate()`. After successfully connecting to EmoEngine, a base profile, which contains initial settings for all detections, may be obtained via the API call `IEE_GetBaseProfile()`.

This function is not required in order to interact with the EmoEngine profile mechanism – **a new user profile with all appropriate default settings is automatically created when a user connects to EmoEngine and the IEE_UserAdded event is generated** - it is, however, useful for certain types of applications that wish to maintain valid profile data for each saved user.

It is much more useful to be able to retrieve the custom settings of an active user. Listing 8 demonstrates how to retrieve this data from EmoEngine.

```
if (IEE_GetUserProfile(userID, eProfile) != EDK_OK) {
      // error in arguments...
}
// Determine the size of a buffer to store the user's profile data
unsigned int profileSize;
if (IEE_GetUserProfileSize(eProfile, &profileSize) != EDK_OK) {
   // you didn't check the return value above…
}
// Copy the content of profile byte stream into local buffer
unsigned char* profileBuffer = new unsigned char[profileSize];
int result;
result=IEE_GetUserProfileBytes(eProfile, profileBuffer, profileSize);
```

Listing 8 ***Get the profile for a particular user***

`IEE_GetUserProfile()` is used to get the profile in use for a particular user. This function requires a valid user ID and an `EmoEngineEventHandle` previously obtained via a call to `IEE_ProfileEventCreate()`. Once again, the return value should always be checked. If successful, an internal representation of the user's profile will be attached to the `EmoEngineEventHandle` and a serialized, binary representation can be retrieved by using the `IEE_GetUserProfileSize()` and `IEE_EngineGetUserProfileBytes()` functions, as illustrated above.

The application is then free to manage this binary profile data in the manner that best fits its purpose and operating environment. For example, the application programmer may choose to save it to disk, persist it in a database or attach it to another app-specific data structure that holds its own per-user data.

```
unsigned int profileSize = 0;
unsigned char* profileBuf = NULL;

// assign and populate profileBuf and profileSize correctly
...

if (IEE_SetUserProfile(userID, profileBuf, profileSize) != EDK_OK) {
    // error in arguments...
}
```

Listing 9 ***Setting a user profile***

`IEE_SetUserProfile()` is used to dynamically set the profile for a particular user. In Listing 9, the `profileBuf` is a pointer to the buffer of the binary profile and `profileSize` is an integer storing the number of bytes of the buffer. The binary data can be obtained from the base profile if there is no previously saved profile, or if the application wants to return to the default settings. The return value should always be checked to ensure the request has been made successfully.

```
...
IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
IEE_EmoEngineEventGetUserId(eEvent, &userID);
switch (eventType) {
    // New Emotiv device connected
    case IEE_UserAdded:
            ...
            break;

    // Emotiv device disconnected
    case IEE_UserRemoved:
            ...
            break;

    // Handle EmoState update
```

```
    case IEE_EmoStateUpdated:
            ...
            break;

    default:
            break;
}
...
```

Listing 10 ***Managing profiles***

Examples 1 and 2 focused chiefly on the proper handling of the `IEE_EmoStateUpdated` event to accomplish their tasks. Two new event types are required to properly manage EmoEngine profiles in Example 3:

1. `IEE_UserAdded`: Whenever a new Emotiv USB receiver is plugged into the computer, EmoEngine will generate an `IEE_UserAdded` event. In this case, the application should create a mapping between the Emotiv user ID for the new device and any application-specific user identifier. The Emotiv USB receiver provides 4 LEDs that can be used to display a player number that is assigned by the application. After receiving the `IEE_UserAdded` event, the `IEE_SetHardwarePlayerDisplay()` function can be called to provide a visual indication of which receiver is being used by each player in a game.
2. `IEE_UserRemoved`: When an existing Emotiv USB receiver is removed from the host computer, EmoEngine™ will send an `IEE_UserRemoved` event to the application and release internal resources associated with that Emotiv device. The user profile that is coupled with the removed Emotiv EPOC™ will be embedded in the event as well. The developer can retrieve the binary profile using the `IEE_GetUserProfileSize()` and `IEE_GetUserProfileBytes()` functions as described above. The binary profile can be saved onto disc to decrease memory usage, or kept in the memory to minimize the I/O overhead, and can be reused at a later time if the same user reconnects.

1.7 Example 4 – Mental Commands Demo

This example demonstrates how the user's conscious mental intention can be recognized by the Mental Commands detection and used to control the movement of a 3D virtual object. It also shows the steps required to train the Mental Commands to recognize distinct mental actions for an individual user.

The design of the Mental Commands Demo application is quite similar to the Facial Expressions Demo covered in Example 2. In Example 2, Facial Expressions Demo retrieves EmoStates™ from Emotiv EmoEngine™ and uses the EmoState data describing the user's facial expressions to control an external avatar. In this example, information about the Mental Commands mental activity of the users is extracted instead. The output of the Mental Commands detection indicates whether users are mentally engaged in one of the trained Mental Commands actions (pushing, lifting, rotating, etc.) at any given time. Based on the Mental Commands results, corresponding commands are sent to a separate application called `EmoCube` to control the movement of a 3D cube.

Commands are communicated to EmoCube via a UDP network connection. As in Example 2, the network protocol is very simple: an action is communicated as two comma-separated, ASCII-formatted values. The first is the action type returned by `IS_MentalCommandsGetCurrentAction()`, and the other is the action power returned by `IS_MentalCommandsGetCurrentActionPower()`, as shown in Listing 11.

```
void sendMentalCommandsAnimation(SocketClient& sock, EmoStateHandle eState) {
    std::ostringstream os;

    IEE_MentalCommandsAction_t actionType;
    actionType = IS_MentalCommandsGetCurrentAction(eState);
    float    actionPower;
    actionPower = IS_MentalCommandsGetCurrentActionPower(eState);

    os << static_cast<int>(actionType) << ","
       << static_cast<int>(actionPower*100.0f);
    sock.SendBytes(os.str());
}
```

Listing 11 ***Querying EmoState for Mental Commands detection results***

1.7.1 Training for Mental Commands

The Mental Commands detection suite requires a training process in order to recognize when a user is consciously imagining or visualizing one of the supported Mental Commands actions. Unlike the Facial Expressions, there is no universal signature that will work well across multiple individuals. An application creates a trained Mental Commands signature for an individual user by calling the appropriate Mental Commands API functions and correctly handling appropriate EmoEngine events. The training protocol is very similar to that described in Example 2 in order to create a trained signature for Facial Expressions.

To better understand the API calling sequence, an explanation of the Mental Commands detection is required. As with Facial Expressions, it will be useful to first familiarize yourself with the operation of the Mental Commands tab in Emotiv before attempting to use the Mental Commands API functions.

Mental Commands can be configured to recognize and distinguish between up to 4 distinct actions at a given time. New users typically require practice in order to reliably evoke and switch between the mental states used for training each Mental Commands action. As such, it is imperative that a user first masters a single action before enabling two concurrent actions, two actions before three, and so forth.

During the training update process, it is important to maintain the quality of the IEEG signal and the consistency of the mental imagery associated with the action being trained. Users should refrain from moving and should relax their face and neck in order to limit other potential sources of interference with their IEEG signal.

Unlike Facial Expressions, the Mental Commands algorithm does not include a delay after receiving the `MC_START` training command before it starts recording new training data.

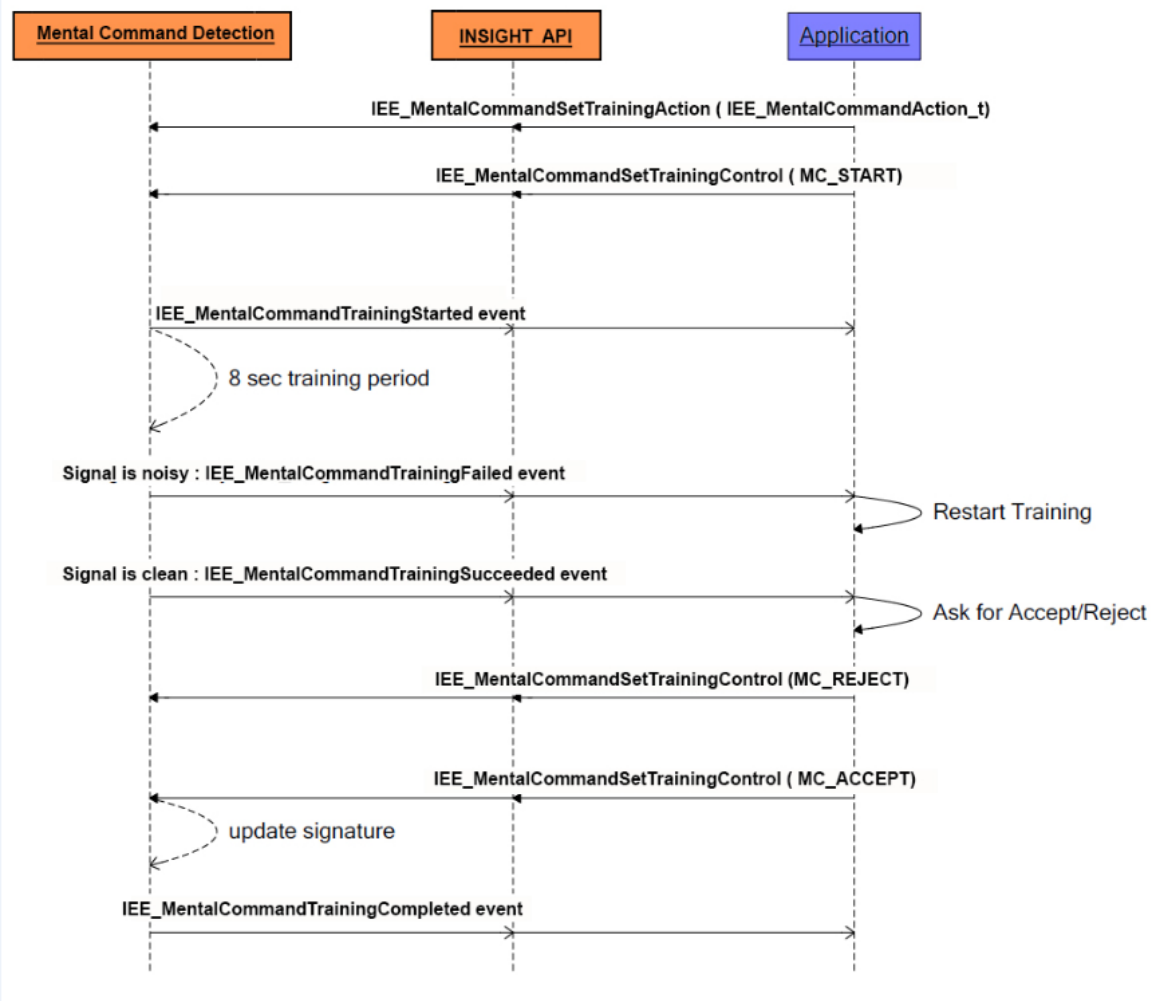


Figure 4 ***Mental Commands training***

The above sequence diagram describes the process of carrying out Mental Commands training on a particular action. The Mental Commands-specific events are declared as enumerated type `IEE_Mental CommandsEvent_t` in i`EDK.h`. Note that this type differs from the `IEE_Event_t` type used by top-level EmoEngine Events. The code snippet in Listing 12 illustrates the procedure for extracting Mental Commands-specific event information from the EmoEngine event.

```
EmoEngineEventHandle IEEvent   = IEE_EmoEngineEventCreate();
if (IEE_EngineGetNextEvent(eEvent) == EDK_OK) {
      IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
      if (eventType == IEE_MentalCommandsEvent) {
      IEE_MentalCommandsEvent_t cEvt =IEE_MentalCommandsEventGetType
                  (eEvent);
                  ...
}
```

Listing 12 ***Extracting Mental Commands event details***

Before the start of a training session, the action type must be first set with the API function `IEE_MentalCommandsSetTrainingAction()`. In i`EmoStateDLL.h`, the enumerated type `IEE_MentalCommandsAction_t` defines all the Mental Commands actions that are currently supported (`MC_PUSH`, `MC_LIFT`, etc.). If an action is not set before the start of training, `MC_NEUTRAL` will be used as the default.

`IEE_MentalCommandsSetTrainingControl()` can then be called with argument `MC_START` to start the training on the target action. In i`EDK.h`, enumerated type `IEE_MentalCommandsTrainingControl_t` defines the control command constants for Mental Commands training. If the training can be started, an `IEE_MentalCommandsTrainingStarted` event will be sent almost immediately. The user should be prompted to visualize or imagine the appropriate action prior to sending the `MC_START command`. The training update will begin after the EmoEngine sends the `IEE_MentalCommandsTrainingStarted` event. This delay will help to avoid training with undesirable IEEG artifacts resulting from transitioning from a "neutral" mental state to the desired mental action state.

After approximately 8 seconds, two possible events will be sent from the EmoEngine™:

`IEE_MentalCommandsTrainingSucceeded`: If the quality of the IEEG signal during the training session was sufficiently good to update the algorithms trained signature, EmoEngine™ will enter a waiting state to confirm the training update, which will be explained below.

`IEE_MentalCommandsTrainingFailed`: If the quality of the IEEG signal during the training session was not good enough to update the trained signature then the Mental Commands training process will be reset automatically, and user should be asked to start the training again.

If the training session succeeded (`IEE_MentalCommandsTrainingSucceeded` was received) then the user should be asked whether to accept or reject the session. The user may wish to reject the training session if he feels that he was unable to evoke or maintain a consistent mental state for the entire duration of the training period. The user's response is then submitted to the EmoEngine through the API call `IEE_MentalCommandsSetTrainingControl()` with argument `MC_ACCEPT` or `MC_REJECT`. If the training is rejected, then the application should wait until it receives the `IEE_MentalCommandsTrainingRejected` event before restarting the training process. If the training is accepted, EmoEngine™ will rebuild the user's trained Mental Command signature, and an `IEE_MentalCommandsTrainingCompleted` event will be sent out once the calibration is done. Note that this signature building process may take up several seconds depending on system resources, the number of actions being trained, and the number of training sessions recorded for each action.

To test the example, launch the Emotiv and the Composer. In the Emotiv select **Connect→To Composer** and accept the default values and then enter a new profile name. Navigate to the \example4\EmoCube folder and launch the EmoCube, enter 20000 as the UDP port and select **Start Server**. Start a new instance of **MentalCommandsDemo**, and observe that when you use the Mental Commands control in the Composer the EmoCube responds accordingly.

Next, experiment with the training commands available in MentalCommandsDemo to better understand the Mental Commands training procedure described above. Listing 13 shows a sample MentalCommandsDemo session that demonstrates how to train.

```
MentalCommandsDemo> set_actions 0 push lift
==> Setting Mental Commands active actions for user 0...

MentalCommandsDemo>
Mental Commands signature for user 0 UPDATED!
Mental CommandsDemo> training_action 0 push
==> Setting Mental Commands training action for user 0 to "push"...

MentalCommandsDemo > training_start 0
==> Start Mental Commands training for user 0...

MentalCommandsDemo >
Mental Commands training for user 0 STARTED!
MentalCommandsDemo >
Mental Commands training for user 0 SUCCEEDED!
Mental CommandsDemo> training_accept 0
==> Accepting Mental Commands training for user 0...

MentalCommandsDemo >
Mental Commands training for user 0 COMPLETED!
Mental CommandsDemo> training_action 0 neutral
==> Setting Mental Commands training action for user 0 to "neutral"...

MentalCommandsDemo > training_start 0
==> Start Mental Commands training for user 0...

MentalCommandsDemo >
Mental Commands training for user 0 STARTED!
MentalCommandsDemo >
Mental Commands training for user 0 SUCCEEDED!
Mental CommandsDemo> training_accept 0
==> Accepting Mental Commands training for user 0...

MentalCommandsDemo >
Mental Commands training for user 0 COMPLETED!
MentalCommandsDemo >
```

Listing 13 ***Training "push" and "neutral" with Mental CommandsDemo***

1.8 Example 5 – IEEG Logger Demo

Before running this example, make sure you already acquired Premium License and activate it using activation tool or using License Activation tool. See 1.17 Example 14- License Activation tool

This example demonstrates how to extract live IEEG data using the EmoEngineTM in C++. Data is read from the headset and sent to an output file for later analysis.

The example starts in the same manner as the earlier examples (see Listing 1 & 2, Section 5.4). A connection is made to the EmoEngine through a call to IEE_EngineConnect(), or to Composer through a call to IEE_EngineRemoteConnect(). The EmoEngine event handlers and EmoState Buffers are also created as before.

```
        float secs                                                    = 1;
        …
    DataHandle hData = IEE_DataCreate();
        DataSetBufferSizeInSec(secs);

        std::cout << "Buffer size in secs:" << secs << std::endl;
        …==> Setting Mental Commands active actions for user 0...
```

Listing 14 ***Access to IEEG data***

Access to IEEG measurements requires the creation of a DataHandle, a handle that is used to provide access to the underlying data. This handle is initialized with a call to IEE_DataCreate(). During the measurement process, EmoEngine will maintain a data buffer of sampled data, measured in seconds. This data buffer must be initialized with a call to DataSetBufferSizeInSec(…), prior to collecting any data.

```
    while (…)

        state = IEE_EngineGetNextEvent(eEvent);

        if (state == EDK_OK) {

            IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
            IEE_EmoEngineEventGetUserId(eEvent, &userID);

                // Log the EmoState if it has been updated

                if (eventType == IEE_UserAdded) {


                    IEE_DataAcquisitionEnable(userID,true);
                    readytocollect = true;
                }
            }
```

Listing 15 ***Start Acquiring Data***

When the connection to EmoEngine is first made via IEE_EngineConnect(), the engine will not have registered a valid user. The trigger for this registration is an IEE_UserAdded event, which is raised shortly after the connection is made. Once the user is registered, it is possible to enable data acquisition via a call to DataAcquisitionEnable. With this enabled, EmoEngine will start collecting IEEG for the user, storing it in the internal EmoEngine sample buffer. Note that the developer's application should access the IEEG data at a rate that will ensure the sample buffer is not overrun.

```
    if (readytocollect)

…
```

```
DataUpdateHandle (0, hData);

unsigned int nSamplesTaken=0;
DataGetNumberOfSample(hData,&nSamplesTaken);

if (nSamplesTaken != 0)
…
        double* data = new double[nSamplesTaken];
        IEE_DataGet(hData, targetChannelList[i], data, nSamplesTaken);
        delete[] data;
```

Listing 16 ***Acquiring Data***

To initiate retrieval of the latest IEEG buffered data, a call is made to DataUpdateHandle(). When this function is processed, EmoEngine will ready the latest buffered data for access via the hData handle. All data captured since the last call to DataUpdateHandle will be retrieved. Place a call to DataGetNumberOfSample() to establish how much buffered data is currently available. The number of samples can be used to set up a buffer for retrieval into your application as shown.

Finally, to transfer the data into a buffer in our application, we call the IEE_DataGet function. To retrieve the buffer, we need to choose from one of the available data channels:

IED_COUNTER, IED_GYROSCOPEX, IED_GYROSCOPEZ, IED_GYROSCOPEX, IED_GYROSCOPEY, IED_T7, IED_ACCX, IED_Pz, IED_ACCY, IED_ACCZ, IED_T8, IED_MAGY, IED_MAGZ, IED_MAGX, IED_MAGZ, IED_GYROX, IED_GYROY, IED_TIMESTAMP, IED_FUNC_ID, IED_FUNC_VALUE, IED_MARKER, IED_SYNC_SIGNAL

For example, to retrieve the first sample of data held in the sensor AF3, place a call to IEE_DataGet as follows:

```
IEE_DataGet(hData, ED_AF3, databuffer, 1);
```

You may retrieve all the samples held in the buffer using the bufferSizeInSample parameter.

Finally, we need to ensure correct clean up by disconnecting from the EmoEngine and free all associated memory.

```
IEE_EngineDisconnect();

IEE_EmoStateFree(eState);

IEE_EmoEngineEventFree(eEvent);
```

1.9 Example 6 – Performance Metrics Demo

Before running this example, make sure you already acquired Premium License and activate it using activation tool or using License Activation tool. See 1.17 Example 14- License Activation tool

Performance MetricsDemo allows log score of Performance Metrics(including raw score and scaled score) in csv file format.

The program runs with command line syntax: EmoStateLogger [log_file_name], log_file_name is set by the user.

```
if (argc != 2)
{
throw std::exception("Please supply the log file name.\nUsage:
EmoStateLogger [log_file_name].");
}
```

Listing 17 ***Creat log_file_name***

The example starts in the same manner as the earlier examples (see Listing 1 & 2, Section 5.4). A connection is made to the EmoEngine through a call to IEE_EngineConnect(), or to Composer through a call to IEE_EngineRemoteConnect().

```
std::cout << "=================================================" << std::endl;
std::cout << "Example to show how to log the EmoState from EmoEngine/Composer ." <<
std::endl;
std::cout << "=================================================" << std::endl;
std::cout << "Press '1' to start and connect to the EmoEngine      " << std::endl;
std::cout << "Press '2' to connect to the Composer                 " << std::endl;
std::cout << ">> ";
std::getline(std::cin, input, '\n');
option = atoi(input.c_str());

switch (option) {
        case 1:
            {
            if (IEE_EngineConnect() != EDK_OK) {
                throw std::exception("Emotiv Engine start up failed.");
            }
            break;
            }
        case 2:
            {
                std::cout << "Target IP of Composer ? [127.0.0.1] ";
                std::getline(std::cin, input, '\n');
                if (input.empty()) {
                                        input = std::string("127.0.0.1");
                                        }
                if (IEE_EngineRemoteConnect(input.c_str(), composerPort) != EDK_OK)
           {
                        std::string errMsg = "Cannot connect to Composer  on [" + input +
                "]";
                        throw std::exception(errMsg.c_str());
                        }
                        break;
                        }
        default:
                throw std::exception("Invalid option...");
```

```
                    break;
                    }


std::cout << "Start receiving Performance MetricsScore! Press any key to stop logging...\n" <<
std::endl;
std::ofstream ofs(argv[1]);
//std::ofstream ofs("test.csv");
bool writeHeader = true;

while (!_kbhit()) {

    state = IEE_EngineGetNextEvent(eEvent);

        // New event needs to be handled
        if (state == EDK_OK) {
                IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
                IEE_EmoEngineEventGetUserId(eEvent, &userID);

        // Log the EmoState if it has been updated
        if (eventType == IEE_EmoStateUpdated) {

                IEE_EmoEngineEventGetEmoState(eEvent, eState);
                const float timestamp = IS_GetTimeFromStart(eState);
         printf("%10.3fs : New Performance Metricsscore from user %d ...\r", timestamp, userID);

                logPerformanceMetricScore(ofs, userID, eState, writeHeader);
                writeHeader = false;
                }
                }
        else if (state != EDK_NO_EVENT) {
                std::cout << "Internal error in Emotiv Engine!" << std::endl;
        break;
                }

        Sleep(1);
                }

        ofs.close();
        }
        catch (const std::exception& e) {
                std::cerr << e.what() << std::endl;
                std::cout << "Press any key to exit..." << std::endl;
                getchar();
        }
```

Listing 18 ***Connect to EmoEngine and Composer***

Log file log.csv has columns as time (time from the beginning of the log), user id, raw score, min, max, scaled score of the PerformanceMetric (Stress, Engagement, Relaxation, Excitement

```
	// Create the top header
	if (withHeader) {
		os << "Time,";
		os << "UserID,";
		os << "Stress raw score,";
		os << "Stress min score,";
		os << "Stress max score,";
		os << "Stress scaled score,";
		os << "Engagement boredom raw score,";
		os << "Engagement boredom min score,";
		os << "Engagement boredom max score,";
		os << "Engagement boredom scaled score,";
		os << "Relaxation raw score,";
		os << "Relaxation min score,";
		os << "Relaxation max score,";
		os << "Relaxation scaled score,";
		os << "Excitement raw score,";
		os << "Excitement min score,";
		os << "Excitement max score,";
		os << "Excitement scaled score,";
		os << std::endl;
	}
	// Log the time stamp and user ID
	os << IS_GetTimeFromStart(eState) << ",";
	os << userID << ",";
	// PerformanceMetric results
	double rawScore=0;
	double minScale=0;
	double maxScale=0;
	double scaledScore=0;
	IS_PerformanceMetricGetStressModelParams(eState,&rawScore,&minScale,&maxScale);
	os << rawScore << ",";
	os << minScale << ",";
	os << maxScale << ",";
	if (minScale==maxScale)
	{
		os << "undefined" << ",";
	}
	else{
		CaculateScale(rawScore,maxScale, minScale,scaledScore);
		os << scaledScore << ",";
	}
	IS_PerformanceMetricGetEngagementBoredomModelParams(eState,&rawScore,&minSc
ale,&maxScale);
	os << rawScore << ",";
	os << minScale << ",";
	os << maxScale << ",";
	if (minScale==maxScale)
	{
		os << "undefined" << ",";
	}
	else{
		CaculateScale(rawScore,maxScale, minScale,scaledScore);
		os << scaledScore << ",";
```

```
        }
        IS_PerformanceMetricGetRelaxationModelParams(eState,&rawScore,&minScale,&maxSc
ale);
        os << rawScore << ",";
        os << minScale << ",";
        os << maxScale << ",";
        if (minScale==maxScale)
        {
                os << "undefined" << ",";
        }
        else{
                CaculateScale(rawScore,maxScale, minScale,scaledScore);
                os << scaledScore << ",";
        }
        IS_PerformanceMetricGetInstantaneousExcitementModelParams(eState,&rawScore,&mi
nScale,&maxScale);
        os << rawScore << ",";
        os << minScale << ",";
        os << maxScale << ",";
        if (minScale==maxScale)
        {
                os << "undefined" << ",";
        }
        else{
                CaculateScale(rawScore,maxScale, minScale,scaledScore);
                os << scaledScore << ",";
        }
        os << std::endl;
}
void CaculateScale (double& rawScore, double& maxScale, double& minScale, double&
scaledScore){

        if (rawScore<minScale)
        {
                scaledScore =0;
        }else if (rawScore>maxScale)
        {
                scaledScore = 1;
        }
        else{
                scaledScore = (rawScore-minScale)/(maxScale-minScale);
        }
}
```

Listing 19 ***Log score to csv file***

Finally, we need to ensure correct clean up by disconnecting from the EmoEngine and free all associated memory.

```
IEE_EngineDisconnect();

IEE_EmoStateFree(eState);

IEE_EmoEngineEventFree(eEvent);
```

1.10 Example 7 – EmoState and IEEGLogger

This example demonstrates the use of the core Emotiv API functions described in Sections 1.2 and 1.3. It logs all Emotiv detection results for the attached users after successfully establishing a connection to Emotiv EmoEngine™ or Composer ™.

Please note that this examples only works with the SDK versions that allow raw IEEG access (Research, Education and Enterprise Plus).

The data is recorded in IEEG_Data.csv files and PerformanceMetrics.csv, they put in the folder .. \ bin \.

```
std::cout << "Start receiving IEEG Data and Performance Metricsdata! Press any key to stop
logging...\n" << std::endl;
std::ofstream ofs("../bin/IEEG_Data.csv",std::ios::trunc);
ofs << header << std::endl;
std::ofstream ofs2("../bin/PerformanceMetrics.csv",std::ios::trunc);
ofs2 << PerformanceMetricSuitesName << std::endl;

DataHandle hData = IEE_DataCreate();
IEE_DataSetBufferSizeInSec(secs);

std::cout << "Buffer size in secs:" << secs << std::endl;
```

Listing 20 ***Log score to IEEG_Data.csv and PerformanceMetrics.csv***

EEG_Data.csv file stores channels :

IED_COUNTER,
IED_GYROSCOPEX, IED_GYROSCOPEZ, IED_GYROSCOPEX, IED_GYROSCOPEY, IED_T7,
IED_ACCX, IED_Pz, IED_ACCY, IED_ACCZ, IED_T8,
IED_MAGY, IED_MAGZ, IED_MAGX, IED_MAGZ, IED_GYROX, IED_GYROY,IED_TIMESTAMP,
IED_FUNC_ID, IED_FUNC_VALUE, IED_MARKER, IED_SYNC_SIGNAL

PerformanceMetric.csv file stores : Engagement, Stress , Relaxation, Excitement

```
while (!_kbhit()) {

        state = IEE_EngineGetNextEvent(eEvent);
        IEE_Event_t eventType;

        if (state == EDK_OK) {

          eventType = IEE_EmoEngineEventGetType(eEvent);
          IEE_EmoEngineEventGetUserId(eEvent, &userID);
          IEE_EmoEngineEventGetEmoState(eEvent, eState);

      // Log the EmoState if it has been updated
          if (eventType == IEE_UserAdded) {
```

```
                std::cout << "User added";
                IEE_DataAcquisitionEnable(userID,true);
                readytocollect = true;
            }


        if (readytocollect && (eventType == IEE_EmoStateUpdated)) {

                IEE_DataUpdateHandle(0, hData);

                unsigned int nSamplesTaken=0;
                IEE_DataGetNumberOfSample(hData,&nSamplesTaken);

                std::cout << "Updated " << nSamplesTaken << std::endl;


                if (nSamplesTaken != 0  ) {

              double* data = new double[nSamplesTaken];
                   for (int sampleIdx=0 ; sampleIdx<(int)nSamplesTaken ; ++      sampleIdx) {
                      for (int i = 0 ;i<sizeof(targetChannelList) /sizeof(IEE_DataChannel_t) ; i++)
            {

                            IEE_DataGet(hData, targetChannelList[i], data, nSamplesTaken);
                            ofs << data[sampleIdx] << ",";
                            }
                            ofs << std::endl;
                }
                delete[] data;
            }

        float affEngegement =IS_PerformanceMetricGetEngagementBoredomScore(eState);
        float affFrus = IS_PerformanceMetricGetStressScore(eState);
        float affMed = IS_PerformanceMetricGetRelaxationScore(eState);
        float affExcitement=IS_PerformanceMetricGetInstantaneousExcitementScore
       (eState);
        printf("Engagement: %f, Stress: %f, ...\n",affEngegement,affFrus);
        ofs2 <<affEngegement<<","<<affFrus<<","<<affMed<<","<<affExcitement<<","
                                        <<std::endl;

        }
        }
        Sleep(100);
}

        ofs.close();
        ofs2.close();
        IEE_DataFree(hData);
```

Listing 21 ***Write data channels and score***

Before the end of the program, `IEE_EngineDisconnect()` is called to terminate the connection with the EmoEngine and free up resources associated with the connection. The user

should also call `IEE_EmoStateFree()` and `IEE_EmoEngineEventFree()` to free up memory allocated for the EmoState buffer and `EmoEngineEventHandle`

1.11 Example 8 – Gyro Data

Gyro data example allows built-in 2-axis gyroscope position.

Simply turn your head from left to right, up and down. You will also notice the red indicator dot move in accordance with the movement of your head/gyroscope.

```
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glPushMatrix();

  glColor3f(1.0,1.0,1.0);
  drawCircle(800,100);
  glColor3f(0.0,0.0,1.0);
  drawCircle(maxRadius-4000,800);
  glColor3f(0.0,1.0,1.0);
  drawCircle(maxRadius,1000);


  glColor3f(1.0, 0.0, 0.0);
  glRectf(currX-400.0, currY-400.0, currX+400.0, currY+400.0);

  glPopMatrix();
  glutSwapBuffers();
}

void changeXY(int x) // x = 0 : idle
{
        if( currX >0 )
        {
                float temp = currY/currX;
                currX -= incOrDec;
                currY = temp*currX;
        }
        else if( currX < 0)
        {
                float temp = currY/currX;
                currX += incOrDec;
                currY = temp*currX;
        }
        else
        {
                if( currY > 0 ) currY -= incOrDec;
                else if( currY <0 ) currY += incOrDec;
        }
        if( x == 0)
        {
                if( (abs(currX) <= incOrDec) && (abs(currY) <= incOrDec))
                {
```

```
                        xmax = 0;
                        ymax = 0;
                }
                else
                {
                        xmax = currX;
                        ymax = currY;
                }
        }
        else
        {
                if( (abs(currX) <= incOrDec) && (abs(currY) <= incOrDec))
                {
                        xmax = 0;
                        ymax = 0;
                }
        }
}


void updateDisplay(void)
{
  int gyroX = 0,gyroY = 0;
  IEE_HeadsetGetGyroDelta(0,&gyroX,&gyroY);
  xmax += gyroX;
  ymax += gyroY;

  if( outOfBound )
  {
        if( preX != gyroX && preY != gyroY )
        {
                xmax = currX;
                ymax = currY;
        }
  }

  double val = sqrt((float)(xmax*xmax + ymax*ymax));

   std::cout <<"xmax : " << xmax <<" ; ymax : " << ymax << std::endl;


  if( val >= maxRadius )
  {
        changeXY(1);
        outOfBound = true;
        preX = gyroX;
        preY = gyroY;
  }
  else
  {
        outOfBound = false;
                if(oldXVal == gyroX && oldYVal == gyroY)
                {
                        ++count;
                        if( count > 10 )
```

```
                        {
                                changeXY(0);
                        }
                }
                else
                {
                        count = 0;
                        currX = xmax;
                        currY = ymax;
                        oldXVal = gyroX;
                        oldYVal = gyroY;
                }
  }
  Sleep(15);
  glutPostRedisplay();
}
void reshape(int w, int h)
{
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-50000.0, 50000.0, -50000.0, 50000.0, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
  switch (button) {
    case GLUT_LEFT_BUTTON:
      if (state == GLUT_DOWN)
        glutIdleFunc(updateDisplay);
      break;
    case GLUT_MIDDLE_BUTTON:
      if (state == GLUT_DOWN)
        glutIdleFunc(NULL);
      break;
    default:
      break;
  }
}
/*
 *  Request double buffer display mode.
 *  Register mouse input callback functions
 */
int main(int argc, char** argv)
{
  EmoEngineEventHandle hEvent = IEE_EmoEngineEventCreate();
  EmoStateHandle eState = IEE_EmoStateCreate();
  unsigned int userID = -1;
  IEE_EngineConnect();
  if(oneTime)
  {
    printf("Start after 8 seconds\n");
          Sleep(8000);
          oneTime = false;
```

```
}

globalElapsed = GetTickCount();

glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize (650, 650);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutIdleFunc(updateDisplay);
glutMainLoop();
```

Listing 22 ***Gyro Data***

Before the end of the program, IEE_EngineDisconnect() is called to terminate the connection with the EmoEngine and free up resources associated with the connection. The user should also call IEE_EmoStateFree() and IEE_EmoEngineEventFree() to free up memory allocated for the EmoState buffer and EmoEngineEventHandle

1.12 Example 9 – Multi Dongle Connection

This example captures event when you plug or unplug dongle .

Every time you plug or unplug dongle, there is a notice that dongle ID is added or removed

```
int main(int argc,char** argv[])
{
        EmoEngineEventHandle hEvent = IEE_EmoEngineEventCreate();
        EmoStateHandle eState = IEE_EmoStateCreate();
        unsigned int userID = -1;
        list<int> listUser;

        if( IEE_EngineConnect() == EDK_OK )
        {
                while(!_kbhit())
                {
                int state = IEE_EngineGetNextEvent(hEvent);
                if( state == EDK_OK )
                {
                        IEE_Event_t eventType = IEE_EmoEngineEventGetType(hEvent);

                        IEE_EmoEngineEventGetUserId(hEvent, &userID);
                        if(userID==-1)
                                continue;

                        if(eventType == IEE_EmoStateUpdated  )
                        {
                        // Copies an EmoState returned with a IEE_EmoStateUpdate event to
                        memory referenced by an EmoStateHandle.
                        if(IEE_EmoEngineEventGetEmoState(hEvent,eState)==EDK_OK)
```

```
                        {

                                if(IEE_GetUserProfile(userID,hEvent)==EDK_OK)
                                {
                                //PerformanceMetric score, short term excitement

                                cout <<"userID: " << userID  <<endl;
                                cout <<"    PerformanceMetric excitement score: " <<
                        IS_PerformanceMetricGetExcitementShortTermScore (eState) << endl;
                                cout <<"    Facial Expressions smile extent : " <<
                                IS_FacialExpressivGetSmileExtent(eState) <<endl;

                                                }
                                        }

                                }
                                // userremoved event
                                else if( eventType == IEE_UserRemoved )
                                {
                                        cout <<"user ID: "<<userID<<" have removed" <<
        endl;
                                        listUser.remove(userID);
                                }
                                // useradded event
                                else if(eventType == IEE_UserAdded)
                                {
                                        listUser.push_back(userID);
                                        cout <<"user ID: "<<userID<<" have added" << endl;
                                }
                                userID=-1;
                        }
                }
        }
```

Listing 23 ***Multi Dongle Connection***

Before the end of the program, IEE_EngineDisconnect() is called to terminate the connection with the EmoEngine and free up resources associated with the connection. The user should also call IEE_EmoStateFree() and IEE_EmoEngineEventFree() to free up memory allocated for the EmoState buffer and EmoEngineEventHandle

1.13 Example 10 – Multi Dongle IEEGLogger

This example logs IEEG data from two headset to data1.csv và data2.csv file in folder "..\bin\".

```
// Create some structures to hold the data
        EmoEngineEventHandle IEEvent = IEE_EmoEngineEventCreate();
        EmoStateHandle eState = IEE_EmoStateCreate();

        std::ofstream ofs1("../bin/data1.csv",std::ios::trunc);
        ofs1 << header << std::endl;
        std::ofstream ofs2("../bin/data2.csv",std::ios::trunc);
        ofs2 << header << std::endl;
```

Listing 24 ***Creat data1.csv and data2.csv for Multi Dongle IEEGLogger***

Please note that this examples only works with the SDK versions that allow raw IEEG access (Research, Education and Enterprise Plus).

Data1.csv or data2.csv file stores channels : IED_COUNTER, IED_GYROSCOPEX, IED_GYROSCOPEZ, IED_GYROSCOPEX, IED_GYROSCOPEY, IED_T7, IED_ACCX, IED_Pz, IED_ACCY, IED_ACCZ, IED_T8,

IED_MAGY, IED_MAGZ, IED_MAGX, IED_MAGZ, IED_GYROX, IED_GYROY, IED_TIMESTAMP, IED_FUNC_ID, IED_FUNC_VALUE, IED_MARKER, IED_SYNC_SIGNAL

```
// Make sure we're connect
if( IEE_EngineConnect() == EDK_OK )
{

// Create the data holder
        DataHandle eData = IEE_DataCreate();
        IEE_DataSetBufferSizeInSec(secs);

// Let them know about it
        std::cout << "Buffer size in secs:" << secs << std::endl;

// How many samples per file?
        int samplIS_per_file = 384;                     // 3 seconds
// Presumably this will fail when we no longer
//      receive data...
        while(!_kbhit())
                {
                // Grab the next event.
                // We seem to mainly care about user adds and removes
                int state = IEE_EngineGetNextEvent(eEvent);
                if( state == EDK_OK )
                {
                // Grab some info about the event
                IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent); // same

                IEE_EmoEngineEventGetUserId(eEvent, &userID); // same

                // Do nothing if no user...
                if(userID==-1) {
                        continue;
                                }


                // Add the user to the list, if necessary
                if (eventType == IEE_UserAdded)
                {
```

```
                std::cout << "User added: " << userID << endl;
                IEE_DataAcquisitionEnable(userID,true);
                userList[numUsers++] = userID;

        // Check
        if (numUsers > 2)
        {
                throw std::exception("Too many users!");
        }
        }
        else if (eventType == IEE_UserRemoved)
        {
                cout << "User removed: " << userID << endl;
                if (userList[0] == userID)
                        {
                        userList[0] = userList[1];
                        userList[1] = -1;
                        numUsers--;
                        }
                else if (userList[1] == userID)
                        {
                        userList[1] = -1;
                        numUsers--;
                                }
                        }

                        // Might be ready to get going.
                        if (numUsers == 2) {
                                readytocollect = true;
                        }
        else {
                                readytocollect = false;
                        }
                    }

                //IEE_DataUpdateHandle(userID, eData);

                // If we've got both, then start collecting
                if (readytocollect && (state==EDK_OK))
                {
                int check = IEE_DataUpdateHandle(userID, eData);
                unsigned int nSamplesTaken=0;
                IEE_DataGetNumberOfSample(eData,&nSamplesTaken);

                if( userID == 0 )
                {
                    if( nSamplesTaken != 0)
                        {
                        IsHeadset1On = true;
                        if( onetime) { write = userID; onetime = false; }
                        for (int c = 0 ; c < sizeof(targetChannelList)
                        /sizeof(IEE_DataChannel_t) ; c++)
                        {
                        data1[c] = new double[nSamplesTaken];
        IEE_DataGet(eData, targetChannelList[c], data1[c], nSamplesTaken);
```

```
                                numberOfSample1 = nSamplesTaken;
                                }
                                }
                        else IsHeadset1On = false;
                                }

                        if( userID == 1  )
                        {
                                if(nSamplesTaken != 0)
                                {
                                IsHeadset2On = true;
                                if( onetime) { write = userID; onetime = false; }
                                for (int c = 0 ; c < sizeof(targetChannelList)/
                               sizeof(IEE_DataChannel_t) ; c++)
                                {
                                data2[c] = new double[nSamplesTaken];
                                IEE_DataGet(eData, targetChannelList[c], data2[c],
                               nSamplesTaken);
                                numberOfSample2 = nSamplesTaken;
                                }
                                }
                        else
                                IsHeadset2On = false;

                                }

        if( IsHeadset1On && IsHeadset2On)
                        {
                        cout <<"Update " << 0 <<" : " << numberOfSample1 << endl;
                        for (int c = 0 ; c < numberOfSample1  ; c++)
                        {
        for (int i = 0 ; i<sizeof(targetChannelList)/ sizeof(IEE_DataChannel_t) ; i++)
                        {
                                ofs1 << data1[i][c] <<",";
                        }
                                ofs1 << std::endl;
                        //delete data1[c];
                        }
                        cout <<"Update " << 1 <<" : " << numberOfSample2 << endl;
                        for (int c = 0 ; c < numberOfSample2  ; c++)
                        {
        for (int i = 0 ; i<sizeof(targetChannelList)/ sizeof(IEE_DataChannel_t) ; i++)
                        {
                                ofs2 << data2[i][c] << ",";
                        }
                                ofs2 << std::endl;
                        //delete[] data2[c];
                        }

                        // Don't overload */
                        //Sleep(100);
                        IsHeadset1On = false;
                        IsHeadset2On = false;
                                }
                        }
```

```
                }
        }
        ofs1.close();
        ofs2.close();
```

Listing 25 ***Write data1.csv and data2.csv file***

Finally, we need to ensure correct clean up by disconnecting from the EmoEngine and free all associated memory.

IEE_EngineDisconnect();

IEE_EmoStateFree(eState);

IEE_EmoEngineEventFree(eEvent);

1.14 Example 11 – MultiChannelEEGLogger

```
IEE_DataUpdateHandle(0, hData);
unsigned int nSamplesTaken=0;
IEE_DataGetNumberOfSample(hData,&nSamplesTaken);
std::cout << "Updated " << nSamplesTaken << std::endl;
if (nSamplesTaken != 0) {
    unsigned int channelCount = sizeof(targetChannelList)/
                                sizeof(IEE_DataChannel_t);
    double ** buffer = new double*[channelCount];
    for (int i=0; i<channelCount; i++)
        buffer[i] = new double[nSamplesTaken];
    IEE_DataGetMultiChannels(hData, targetChannelList,
                             channelCount, buffer, nSamplesTaken);
    for (int sampleIdx=0 ; sampleIdx<(int)nSamplesTaken ;
         ++ sampleIdx) {
        for (int i = 0 ; i<sizeof(targetChannelList)/
                           sizeof(IEE_DataChannel_t) ; i++) {
            ofs << buffer[i][sampleIdx] << ",";
        }
        ofs << std::endl;
    }
    for (int i=0; i<channelCount; i++)
        delete buffer[i];
    delete buffer;
}
```

Listing 26 ***Write log IEEG Data from EmoInsightDriver/ Composer***

1.15 Example 12 – ProfileUpload

```
result = EC_Login(userName.c_str(), password.c_str());
if (result != EDK_OK) {
  std::cout << "Your login attempt has failed. The username or password
may be incorrect";#
  ifdef _WIN32
```

```
  _getch();#
  endif
  return result;
}
std::cout << "Logged in as " << userName << std::endl;
result = EC_GetUserDetail( & userCloudID);
if (result != EDK_OK)
  return result;
while (!_kbhit()) {
  state = IEE_EngineGetNextEvent(eEvent);
  if (state == EDK_OK) {
    IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
    IEE_EmoEngineEventGetUserId(eEvent, & engineUserID);
    if (eventType == IEE_UserAdded) {
      std::cout << "User added" << std::endl;
      ready = true;
    }
  }
  if (ready) {
    int getNumberProfile = EC_GetAllProfileName(userCloudID);
    std::cout << "Number of profiles: " << getNumberProfile << "\n";
    for (int i = 0; i < getNumberProfile; i++) {
      std::cout << "Profile Name: " <<
EC_ProfileNameAtIndex(userCloudID, i) << ", ";
      std::cout << "Profile ID: " << EC_ProfileIDAtIndex(userCloudID,
i) << ", ";
      std::cout << "Profile type: " <<
((EC_ProfileTypeAtIndex(userCloudID, i) == profileFileType::TRAINING) ?
"TRAINING" : "EMOKEY") << ", ";
      std::cout << EC_ProfileLastModifiedAtIndex(userCloudID, i) <<
",\r\n";
    }
    switch (option) {
    case 1:
      {
        int profileID = -1;
        result = EC_GetProfileId(userCloudID, profileName.c_str(), &
profileID);
          if (profileID >= 0) {
          std::cout << "Profile with " << profileName << " is existed"
<< std::endl;
          result = EC_UpdateUserProfile(userCloudID, engineUserID,
profileID);
          if (result == EDK_OK) {
            std::cout << "Updating finished";
          } else std::cout << "Updating failed";
        } else {
          result = EC_SaveUserProfile(userCloudID, (int) engineUserID,
profileName.c_str(), TRAINING);
          if (result == EDK_OK) {
            std::cout << "Saving finished";
          } else std::cout << "Saving failed";
        }#
        ifdef _WIN32
        _getch();#endif
        return result;
      }
```

```
    case 2:
      {
          if (getNumberProfile > 0) {
          result = EC_LoadUserProfile(userCloudID, (int) engineUserID,
EC_ProfileIDAtIndex(userCloudID, 0));
          if (result == EDK_OK)
            std::cout << "Loading finished";
          else
            std::cout << "Loading failed";
        }#
        ifdef _WIN32
        _getch();#endif
        return result;
      }
    case 3:
      {
        std::cout << "profile name: " << std::endl;
        std::getline(std::cin, profileName);
        std::cout << "profile path: " << std::endl;
        std::getline(std::cin, pathOfProfile);
        bool overwrite_if_exists = true;
        result = EC_UploadProfileFile(userCloudID, profileName.c_str(),
pathOfProfile.c_str(), TRAINING, overwrite_if_exists);
        if (result == EDK_OK)
          std::cout << "Upload finished";
        else
          std::cout << "Upload failed";#ifdef _WIN32
        _getch();#endif
        return result;
      }
    case 4:
      {
        std::cout << "profile name: " << std::endl;
        std::getline(std::cin, profileName);
        // E:/profile.emu
        std::cout << "profile path to save: " << std::endl;
        std::getline(std::cin, pathOfProfile);
        int profileID = -1;
        result = EC_GetProfileId(userCloudID, profileName.c_str(), &
profileID);
        if (profileID >= 0)
          result = EC_DownloadProfileFile(userCloudID, profileID,
pathOfProfile.c_str());#ifdef _WIN32
        _getch();#endif
        return result;
      }
    default:
      std::cout << "Invalid option...";
      break;
    }
  }#
  ifdef _WIN32
  Sleep(1);#
  endif#
  if __linux__ || __APPLE__
  usleep(10000);#
  endif
```

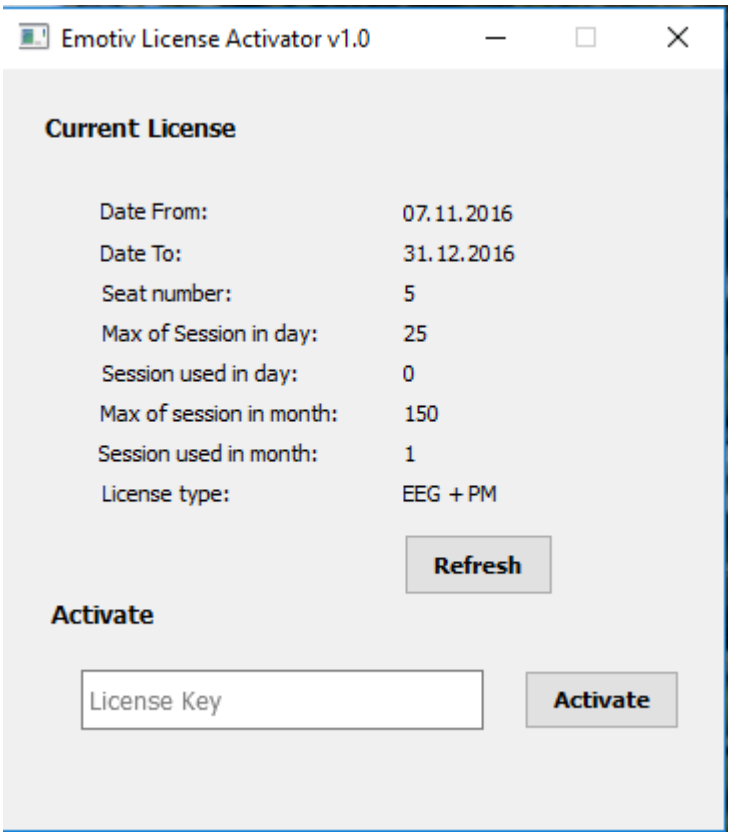
```
}
IEE_EngineDisconnect();
IEE_EmoStateFree(eState);
IEE_EmoEngineEventFree(eEvent);
return 0;
}
```

Listing 27 ***Upload Profile to Cloud***

1.16 Example 13 – License Activation

The simplest way to active license is using Emotiv Tools with License Activator feature, key in your license and press Activate button. License information will be shown as screenshot

This process must be done once on every new computer, after that all Emotiv softwares can run features included in the license. Internet connection is required for activation.



It is also possible to activate the license using Emotiv API, refer to these sample code

```
std::string licenseFilePath = "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx";
int result;
result = IEE_ActivateLicense(licenseFilePath.c_str());
```

Listing 28 ***Activate license***