EMOTIV

GETTING STARTED GUIDE

Version 3.5.0

EMOTIV

Table of Contents

Figures and Tables

Getting started with Emotiv SDK

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 2013 and can be found in the examples_basic\ and examples_extra_prime\ directory of your installation or from our github page.

2. Introduction to the Emotiv API and Emotiv EmoEngine™

The Emotiv API is exposed as an ANSI C interface that is declared in header files contained in include folder 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 EEG 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_."

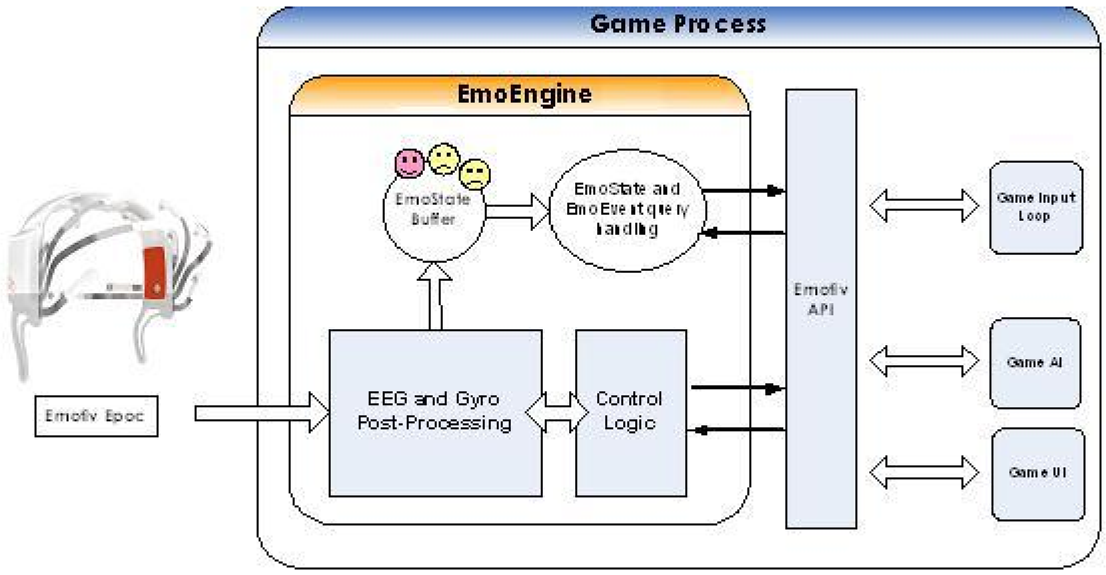

Figure 1 Integrating the EmoEngine and Emotiv EPOC with a videogame

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).

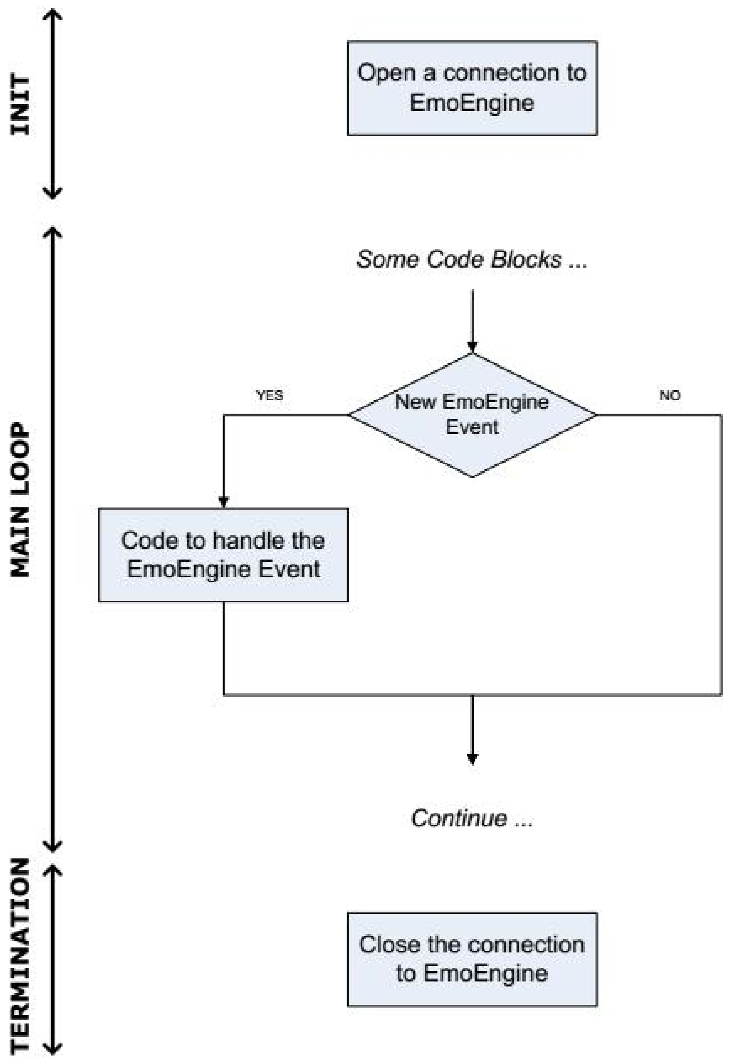
INIT
Open a connection to EmoEngine
MAIN LOOP
Some Code Blocks ...
New EmoEngine Event
YES
NO
Code to handle the EmoEngine Event
Continue ...
TERMINATION
Close the connection to EmoEngine

Figure 2 Using the API to communicate with the EmoEngine

The above figure 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.

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 Appendix 2
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 Appendix 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 XavierComposer, the EmoEngine. XavierComposer 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 XavierComposer 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 XavierComposer . 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 use 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.

4. Examples

4.1 Example 1 – EmoStateLogger

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

```
std::cout << "Press '1' to start and connect to the EmoEngine" << std::endl;
std::cout << "Press '2' to connect to the EmoComposer"<< 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::runtime_error("Emotiv Driver start up failed.");
	}

	break;
}
case 2:
{
	std::cout << "Target IP of EmoComposer? [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 EmoComposer on [" + input + "]";
	  throw std::runtime_error(errMsg.c_str());
	}
	break;
	}
default:
	throw std::runtime_error("Invalid option...");
	break;

      }
}
```

Figure 3 Connect to Engine

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 eEvent        = IEE_EmoEngineEventCreate();
EmoStateHandle eState              = IEE_EmoStateCreate();
unsigned int userID                = 0;
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);
                 std::cout<<timestamp<< ": "<<"New EmoState from user"<<userID;
                logEmoState(ofs, userID, eState, writeHeader);
                writeHeader = false;
         }
  }
  else if (state != EDK_NO_EVENT) {
         std::cout << "Internal error in Emotiv Engine!" << std::endl;
         break;
  }
}
```

Figure 4 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);
```

Figure 5 Disconnecting fom 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.

4.2 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_FacialExpressionAlgo_t upperFaceType =
                            IS_FacialExpressionGetUpperFaceAction(eState);
IEE_FacialExpressionAlgo_t lowerFaceType =
                            IS_FacialExpressionGetLowerFaceAction(eState);

float upperFaceAmp = IS_FacialExpressionGetUpperFaceActionPower(eState);
 float lowerFaceAmp = IS_FacialExpressionGetLowerFaceActionPower(eState);
```

Figure 6 Get Facial expression type and power

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). 0 below excerpts the syntax of some of expressions supported by the protocol

Facial Expressions Actions 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 Blue Avatar 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 detection 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.

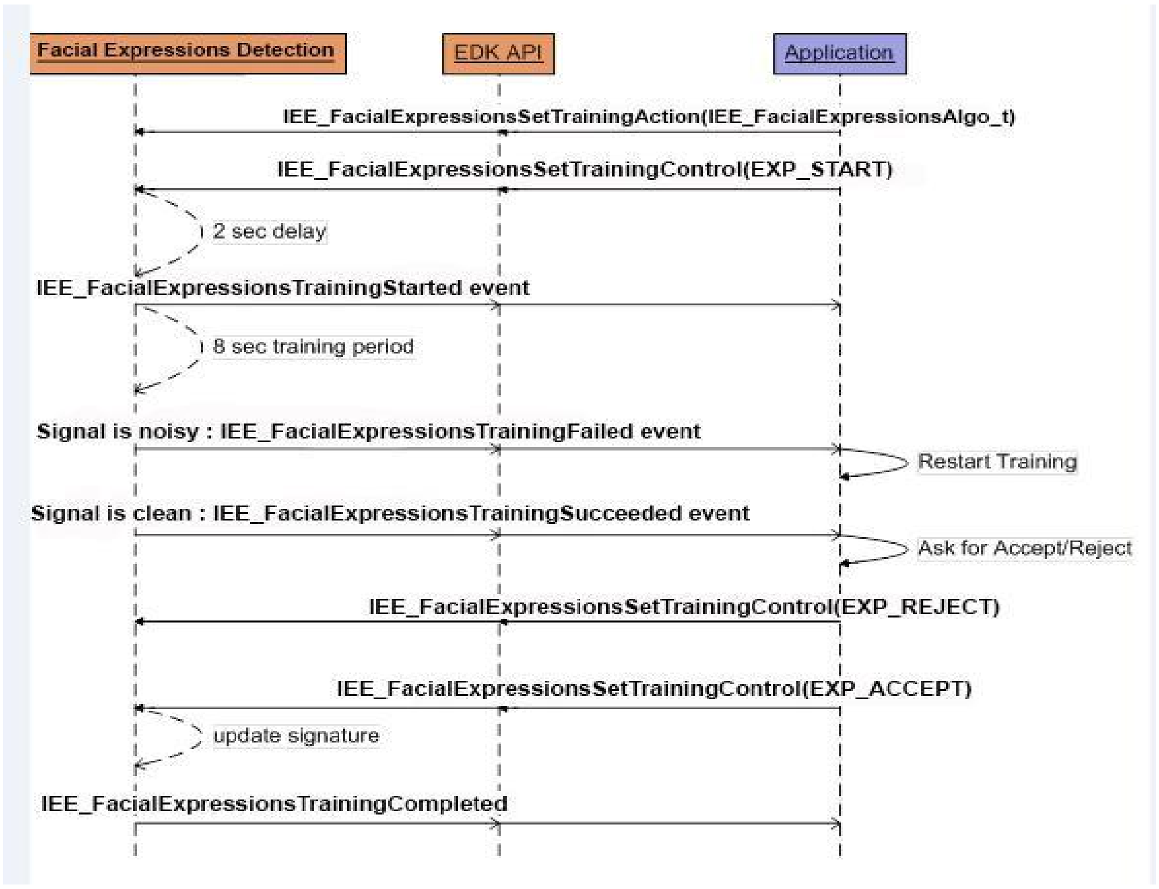
Facial Expressions Detection
EDK API
Application
IEE_FacialExpressionsSetTrainingAction(IEE_FacialExpressionsAlgo_t)
IEE_FacialExpressionsSetTrainingControl(EXP_START)
2 sec delay
IEE_FacialExpressionsTrainingStarted event
8 sec training period
Signal is noisy : IEE_FacialExpressionsTrainingFailed event
Restart Training
Signal is clean : IEE_FacialExpressionsTrainingSucceeded event
Ask for Accept/Reject
IEE_FacialExpressionsSetTrainingControl(EXP_REJECT)
IEE_FacialExpressionsSetTrainingControl(EXP_ACCEPT)
update signature
IEE_FacialExpressionsTrainingCompleted

Figure 7 Facial Expression training flow

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 IEDK.h. Note that this type differs from the IEE_Event_t type used by top-level EmoEngine Events.

```
while (true) {
  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);
         switch (eventType) {
         // if EmoState has been updated Send the FacialExpression animation
         case IEE_EmoStateUpdated:
         {
                IEE_EmoEngineEventGetEmoState(eEvent, eState);
                std::map<unsigned int, SocketClient>::iterator iter;
                iter = socketMap.find(userID);
                if (iter != socketMap.end()) {
                sendFacialExpressionAnimation(iter->second, eState);
                }
                break;
         }
         // Handle FacialExpression training event
         case IEE_FacialExpressionEvent:
         {
                handleFacialExpressionEvent(std::cout, eEvent);
         }
         default:
                break;
         }
  `
```

Figure 8 Extracting Facial Expression 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 examples_basic\C++\FacialExpressionDemo\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. 0 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 4096 added, sending Facial Expressions
animation to localhost:30000...
FacialExpressionsDemo> trained_sig 4096
==> Querying availability of a trained Facial Expressions signature for
user 4096...
A trained Facial Expressions signature is not available for user 4096
FacialExpressionsDemo> training_exp 4096 neutral
==> Setting Facial Expressions training expression for user 4096 to
neutral...
FacialExpressionsDemo> training_start 4096
==> Start Facial Expressions training for user 4096...
FacialExpressionsDemo>
Facial Expressions training for user 4096 STARTED!
FacialExpressionsDemo>Facial Expressions training for user 4096 SUCCEEDED!
FacialExpressionsDemo> training_accept 4096
==> Accepting Facial Expressions training for user 4096...
FacialExpressionsDemo>Facial Expressions training for user 4096 COMPLETED!
FacialExpressionsDemo> training_exp 4096 smile
==> Setting Facial Expressions training expression for user 4096 to
smile...
FacialExpressionsDemo> training_start 4096
==> Start Facial Expressions training for user 4096...
FacialExpressionsDemo>Facial Expressions training for user 4096 STARTED!
FacialExpressionsDemo>Facial Expressions training for user 4096 SUCCEEDED!
FacialExpressionsDemo> training_accept 4096
==> Accepting Facial Expressions training for user 4096...
FacialExpressionsDemo> Facial Expressions training for user 4096
COMPLETED!
FacialExpressionsDemo> trained_sig 4096
==> Querying availability of a trained Facial Expressions signature for
user 4096...
A trained Facial Expressions signature is available for user 4096
FacialExpressionsDemo> set_sig 4096 1
==> Switching to a trained Facial Expressions signature for user 4096..
```

Figure 9 Training "smile" and "neutral" in Facial Expressions Demo

4.3 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.

We have demoed some Profile API in MentalCommand examples. In this session, We have demonstrated other Profile 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);
}
```

Figure 10 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.demonstrates how to retrieve this data from EmoEngine.

```
if (IEE_GetUserProfile(userID, eProfile) != EDK_OK) { }
//Determine the size of a buffer to store the user's profile data unsigned
int profileSize;
if (IEE_GetUserProfileSize(eProfile, &profileSize) != EDK_OK) {}
//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);
```

Figure 11 Get user profile

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) {}
```

Figure 12 Setting a user profile

IEE_SetUserProfile() is used to dynamically set the profile for a particular user. 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.

After logging to Emotiv Cloud, user can work with profile as below functions.

```
  ...
if(EC_Login(userName.c_str(), password.c_str()) != EDK_OK)
  {
        std::cout << "Your login attempt has failed.
        The username or password may be incorrect" << std::endl;
        return;
  }
  //Get user Cloud ID
  if (EC_GetUserDetail(&userCloudID) != EDK_OK) {
  std::cout << "Error: Cannot get detail info, exit " << userName << std::endl;
  return -1;}
  …
void SavingLoadingFunction(int userCloudID, bool save = true)
{
  int getNumberProfile = EC_GetAllProfileName(userCloudID);
  if (save) {    // Save profile to cloud
        int profileID = -1;
        // Get unique Profile ID if ProfileID > 0 --> existed
        EC_GetProfileId(userCloudID, profileName.c_str(), &profileID);
        if (profileID >= 0) {
        std::cout << "Profile with " << profileName << " is existed.";
        if (EC_UpdateUserProfile(userCloudID, userID, profileID) == EDK_OK)
               std::cout << "Updating finished" << std::endl;
        else
               std::cout << "Updating failed" << std::endl;
        }
        else if (EC_SaveUserProfile(userCloudID, userID, profileName.c_str(),
TRAINING) == EDK_OK){
              std::cout << "Saving finished" << std::endl;
        }
         else std::cout << "Saving failed" << std::endl; return;
   } else {
   // Load profile from cloud
   if (getNumberProfile > 0){
   if (EC_LoadUserProfile(userCloudID, userID,EC_ProfileIDAtIndex(userCloudID, 0))
== EDK_OK)
        std::cout << "Loading finished" << std::endl;
    else
        std::cout << "Loading failed" << std::endl;
    }
    return;
```

Figure 13 Save - Update - Load Profile examples

4.4 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()

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

      IEE_MentalCommandAction_t actionType      =
              IS_MentalCommandGetCurrentAction(eState);
      float   actionPower = IS_MentalCommandGetCurrentActionPower(eState);

      os << static_cast<int>(actionType) << ","
         << static_cast<int>(actionPower*100.0f);

       sock.SendBytes(os.str());
}
```

Figure 14 Send MentalCommand Animation through socket

4.4.1. 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.

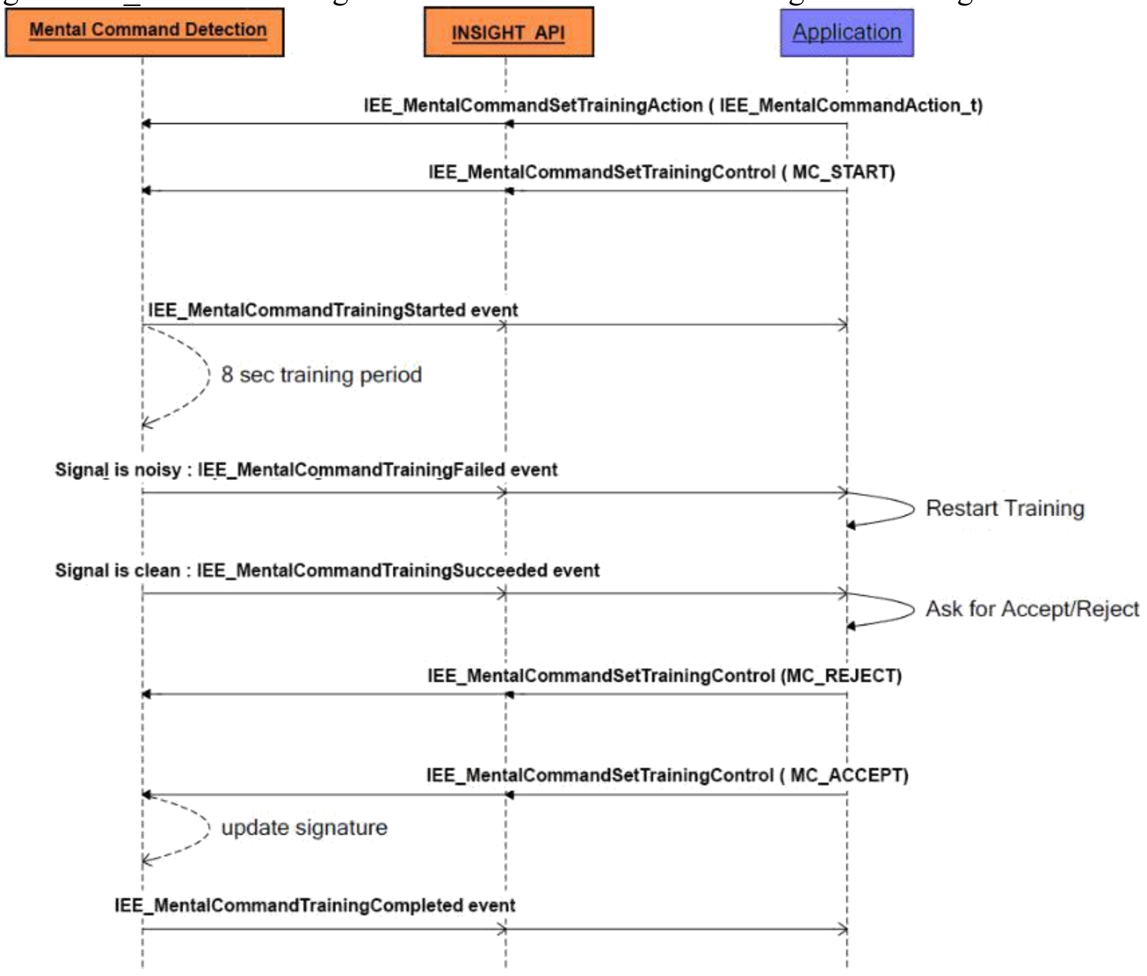
Mental Command Detection
INSIGHT API
Application
IEE_MentalCommandSetTrainingAction (IEE_MentalCommandAction_t)
IEE_MentalCommandSetTrainingControl (MC_START)
IEE_MentalCommandTrainingStarted event
8 sec training period
Signal is noisy : IEE_MentalCommandTrainingFailed event
Restart Training
Signal is clean : IEE_MentalCommandTrainingSucceeded event
Ask for Accept/Reject
IEE_MentalCommandSetTrainingControl (MC_REJECT)
IEE_MentalCommandSetTrainingControl (MC_ACCEPT)
update signature
IEE_MentalCommandTrainingCompleted event

Figure 15 Mental Commands training flow

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 iEDK.h. Note that this type differs from the IEE_Event_t type used by top-level EmoEngine Events. The code snippet in 0 illustrates the procedure for handling Mental Commands-specific event information from the EmoEngine event.

```
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);
        switch (eventType) {
        // New headset connected
        // create a new socket to send the animation
        case IEE_UserAdded:
        {
                std::cout << std::endl << "New user " << userID
                      << " added, sending MentalCommand animation to ";
                socketMap.insert(std::pair<unsigned int, SocketClient>(
                userID, SocketClient(receiverHost, startSendPort, UDP)));
                startSendPort++;
                break;
        }
        // Headset disconnected, remove the existing socket
        case IEE_UserRemoved:
        {
                std::cout << std::endl << "User " << userID << " has been removed.";
                std::map<unsigned int, SocketClient>::iterator iter;
                iter = socketMap.find(userID);
                if (iter != socketMap.end()) {
                socketMap.erase(iter);}
                break;
        }
        // Send the MentalCommand animation if EmoState has been updated
        case IEE_EmoStateUpdated:
        {
                IEE_EmoEngineEventGetEmoState(eEvent, eState);
                std::map<unsigned int, SocketClient>::iterator iter;
                iter = socketMap.find(userID);
                if (iter != socketMap.end()) {
                sendMentalCommandAnimation(iter->second, eState);
        }
        break;
        }
        // Handle MentalCommand training related event
        case IEE_MentalCommandEvent:
        {
                handleMentalCommandEvent(std::cout, eEvent);
                break;
         }
}
```

Figure 16 Handling Mental Command Event

Before the start of a training session, the action type must be first set with the API function IEE_MentalCommandsSetTrainingAction(). In iEmoStateDLL.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 iEDK.h, enumerated type

IEE_MentalCommandsTrainingControl_t defines the control command constants for MentalCommands 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 \examples_basic\C++\MentalCommandDemo\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 which shows a sample MentalCommandsDemo session that demonstrates how to train.

```
MentalCommandsDemo> set_actions 4096 push lift

==> Setting Mental Commands active actions for user 4096...

MentalCommandsDemo> Mental Commands signature for user 4096 UPDATED!
Mental CommandsDemo> training_action 0 push

==> Setting Mental Commands training action for user 4096 to
"push"..

MentalCommandsDemo > training_start 4096

==> Start Mental Commands training for user 4096...

MentalCommandsDemo >Mental Commands training for user 4096 STARTED!

MentalCommandsDemo >Mental Commands training for user 4096
SUCCEEDED!

Mental CommandsDemo> training_accept 4096

==> Accepting Mental Commands training for user 4096...

MentalCommandsDemo >

Mental Commands training for user 4096 COMPLETED! Mental
CommandsDemo> training_action 4096 neutral

==> Setting Mental Commands training action for user 4096 to
"neutral"...

MentalCommandsDemo > training_start 4096

==> Start Mental Commands training for user 4096...

MentalCommandsDemo > Mental Commands training for user 4096 STARTED!

MentalCommandsDemo > Mental Commands training for user 4096
SUCCEEDED!

Mental CommandsDemo> training_accept 4096

==> Accepting Mental Commands training for user 4096...

MentalCommandsDemo > Mental Commands training for user 4096
COMPLETED!
```

Figure 17 Training "push" and "neutral" with MentalCommandDemo

4.5 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 section 4.10- Example 10- 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 above examples . 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();
  IEE_DataSetBufferSizeInSec(secs);

  while (!_kbhit()) {
         state = IEE_EngineGetNextEvent(eEvent);
         if (state == EDK_OK) {
         IEE_Event_t eventType = IEE_EmoEngineEventGetType(eEvent);
         IEE_EmoEngineEventGetUserId(eEvent, &userID);
         if (eventType == IEE_UserAdded) {
                std::cout << "User added" << std::endl;
                IEE_DataAcquisitionEnable(userID, true);
                readytocollect = true;
         }
         }
         if (readytocollect) {
         int result = IEE_DataUpdateHandle(userID, hData);
         if (result != EDK_OK)
                continue;
         unsigned int nSamplesTaken = 0;
         IEE_DataGetNumberOfSample(hData, &nSamplesTaken);

         if (nSamplesTaken != 0) {
                double* data = new double[nSamplesTaken];
         for (int sampleIdx = 0; sampleIdx < (int)nSamplesTaken; ++sampleIdx)
         {
                for (int i = 0; i < sizeof(EpocChannelList) /
sizeof(IEE_DataChannel_t); i++)
                {
                       IEE_DataGet(hData, EpocChannelList[i], data, nSamplesTaken);
                       ofs << data[sampleIdx] << ",";
                }
                ofs << std::endl;
                }
         delete[] data;
         }
         }
  }
```

Figure 18 Get EEG 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.

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.

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);

4.6 Example 6 – Performance Metrics Demo

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.

The example starts in the same manner as the earlier above examples. A connection is made to the EmoEngine through a call to IEE_EngineConnect(), or to Composer through a call to IEE_EngineRemoteConnect().

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

```
  …
std::cout << "Start receiving PerformanceMetric Score! "<< std::endl;

std::ofstream ofs("PerformanceMetricData.csv", std::ios::trunc);
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);
       logPerformanceMetricScore(ofs, userID, eState, writeHeader);
       writeHeader = false;
       }
    }
}
…
void logPerformanceMetricScore(std::ostream& os, unsigned int userID,
         EmoStateHandle eState, bool writeHeader)
  {

  // Create the top header
       if (writeHeader) {
      // Write header
       }
       // PerformanceMetric results
       double rawScore=0, minScale=0, maxScale=0, scaledScore=0;
      // Stress
      IS_PerformanceMetricGetStressModelParams(eState,&rawScore,&minScale,
                                              &maxScale);
      // Engagement
      IS_PerformanceMetricGetEngagementBoredomModelParams(eState,&rawScore,
                                                         &minScale,&maxScale);
      // Relaxation
      IS_PerformanceMetricGetRelaxationModelParams(eState,&rawScore,
                                                  &minScale,&maxScale);
      // Excitement
      IS_PerformanceMetricGetInstantaneousExcitementModelParams(eState,
                                                               &rawScore,&minScale,
                                                               &maxScale);
      // Interest
      IS_PerformanceMetricGetInterestModelParams(eState,&rawScore,
                                                &minScale,&maxScale);
      // Focus
      IS_PerformanceMetricGetFocusModelParams(eState,&rawScore,
                                             &minScale,&maxScale);

  }
```

Figure 19 Get performance metrics data

4.7 Example 7 – Gyro Data

Note: This example just supports 32 bit system. We use GLUT library was was a computer graphics library for OpenGL.
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.

```
#ifdef _WIN32
#include <GL/gl.h>
#include <GL/glu.h>
#include <glut.h>
#elif __APPLE__
#include <OpenGL/gl.h>
#include <OpenGL/glu.h>
#include <GLUT/glut.h>
#include <mach/clock.h>
#include <mach/mach.h>
#else
#include <GL/glut.h>

// Motion channel lists
const IEE_MotionDataChannel_t targetChannelList[] = {
    IMD_COUNTER,
    IMD_GYROX,
    IMD_GYROY,
    IMD_GYROZ,
    IMD_ACCX,
    IMD_ACCY,
    IMD_ACCZ,
    IMD_MAGX,
    IMD_MAGY,
    IMD_MAGZ,
    IMD_TIMESTAMP
};
…
if (IEE_EngineConnect() != EDK_OK) {
            std::cout << "Emotiv Driver start up failed.";
            return -1;
     }

     std::cout << "Please make sure your headset is on and don't move your head.";
// Set Buffer size in second
IEE_MotionDataSetBufferSizeInSec(1);


…
// Set Buffer size in second
state = IEE_EngineGetNextEvent(hEvent);

int gyroX = 0, gyroY = 0;
int err = IEE_HeadsetGetGyroDelta(userID, &gyroX, &gyroY);
if (err == EDK_OK){
     std::cout << "You can move your head now." << std::endl;
     break;
}else if (err == EDK_GYRO_NOT_CALIBRATED){
       std::cout << "Gyro is not calibrated. Please stay still." << std::endl;
       showGyro(true); // Show gyroX, gyroY to console
}else if (err == EDK_CANNOT_ACQUIRE_DATA){
       std::cout << "Cannot acquire data" << std::endl;
       showGyro(true);
}else{
       std::cout << "No headset is connected" << std::endl;
}
```

```
 ...
// Display
 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();
 ofs.close();

 IEE_EngineDisconnect();
 IEE_EmoStateFree(eState);
```

Figure 20 Gyro data examples

4.8 Example 8 – 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.
Please see the below path for example similar to one dongle connection.
/community-sdk/examples_basic/C++/**MultiDongleConnection**/
The value of userID base on connection which connect by dongle or bluetooth.

```
case InputStreamType_t::DONGLE_STREAM:
  userID += 0x1000;//userID start from 4096 and increase 1 for next dongle
case InputStreamType_t::BTLE_STREAM:
  userID += 0x2000;//userID start from 8192 and increase 1 for next connection
```

4.9 Example 9 – MultiChannelEEGLogger

We can get EEG data for each channel as section 4.5 or can get EEG data for multiple channels as below example.

```
    int result = IEE_DataUpdateHandle(userID, hData);
    if (result == EDK_OK)
    {
          unsigned int nSamplesTaken = 0;
          IEE_DataGetNumberOfSample(hData, &nSamplesTaken);
          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];
                 result = IEE_DataGetMultiChannels(hData, targetChannelList,
                              channelCount, buffer, nSamplesTaken);

         }
    }
```

Figure 21 Get EEG data from multiple channels

4.10 Example 10 – 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.

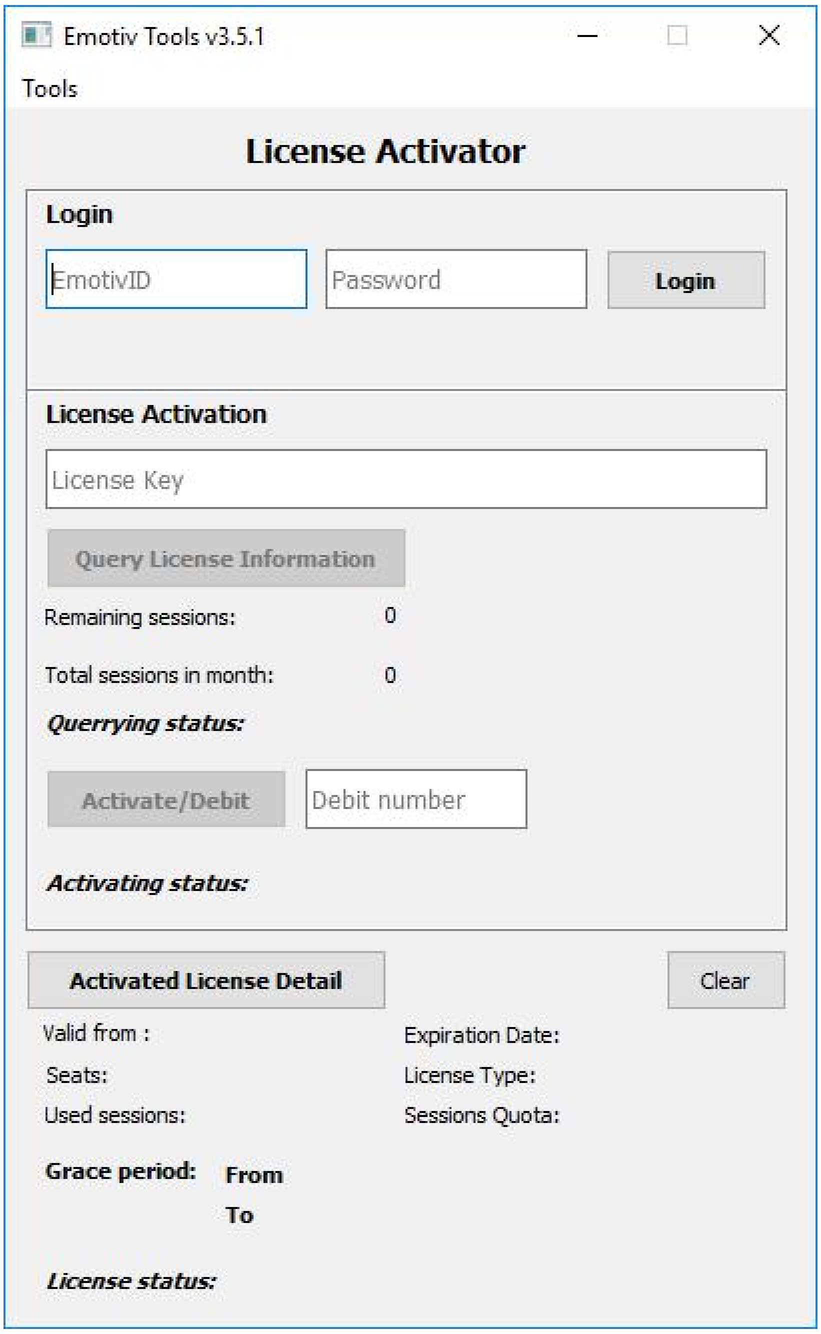

Figure 22 License Activator Tool

It is also possible to activate the license using Emotiv API, refer to ActivateLicense Example

```
…
//Login first
if (EC_Login(userName.c_str(), password.c_str()) != EDK_OK)
{
     std::cout << "Login failed. The username or password may be incorrect";
     return -1;
}
//get Debit info
IEE_DebitInfos_t debitInfos;
debitInfos.remainingSessions = 0;
result = IEE_GetDebitInformation(LICENSE_KEY.c_str(), &debitInfos);
if (result == EDK_OK)
{
       if (debitInfos.total_session_inYear > 0)
       {
              std::cout << "Remaining Sessions in year : "<<
                          debitInfos.remainingSessions;
              std::cout << "the total number of session can be debitable in year :
                          " << debitInfos.total_session_inYear << std::endl;
       }
       else if (debitInfos.total_session_inMonth > 0)
       {
              std::cout << "Remaining Sessions in month : " <<
                                 debitInfos.remainingSessions << std::endl;
              std::cout<<"the total number of session can be debitable in month: "
                          <<     debitInfos.total_session_inMonth << std::endl;
       }
       else {
              std::cout << "Remaining Sessions:unlimitted " << std::endl;
              std::cout << "It is unlimitted license" << std::endl;
       }
}
else
{
       std::cout << std::hex << "GET DEBIT INFORMATION UNSUCCESSFULLY!;
}
…
unsigned int debitNum = 0 // Number of debit , default is 0
std::string const LICENSE_KEY = "";  // Your license bought from Emotiv
result = IEE_AuthorizeLicense(LICENSE_KEY.c_str(), debitNum);
if (!(result == EDK_OK || result == EDK_LICENSE_REGISTERED))
    {
        std::cout << std::hex << "ACTIVE/DEBIT UNSUCCESSFULLY!" << std::endl;
        return result;
    }
…
IEE_LicenseInfos_t licenseInfos;
// We can call this API any time to check current License information
result = IEE_LicenseInformation(&licenseInfos);
std::cout << "From Date          : "<< convertEpochToTime(licenseInfos.date_from);
std::cout << "To   Date          : " << convertEpochToTime(licenseInfos.date_to);
std::cout << "Number of Seats    : " << licenseInfos.seat_count << std::endl;
std::cout << "Total Quotas       : " << licenseInfos.quota << std::endl;
std::cout << "Total used Quotas : " << licenseInfos.usedQuota << std::endl;
std::cout << "Grace period from " <<
convertEpochToTime(licenseInfos.soft_limit_date) << "   To " <<
convertEpochToTime(licenseInfos.hard_limit_date) << std::endl;
```

Figure 23 Authorize License Examples

Appendix 1 Emotiv EmoEngine™ Error Codes

Every time you use a function provided by the API, the value returned indicates the EmoEngine™ status. 0 below shows possible EmoEngine error codes and their meanings.

EmoEngine Error Code	Hex Value	Description
EDK_OK	0x0000	Operation has been carried out successfully.
EDK_UNKNOWN_ERROR	0x0001	An internal fatal error occurred.
EDK_INVALID_PROFILE_ARCHIVE	0x0101	Most likely returned by EE_SetUserProfile() when the content of the supplied buffer is not a valid serialized EmoEngine profile.
EDK_NO_USER_FOR_BASE_PROFILE	0x0102	Returns when trying to query the user ID of a base profile.
EDK_CANNOT_ACQUIRE_DATA	0x0200	Returns when EmoEngine is unable to acquire any signal from Emotiv EPOC™ for processing
EDK_BUFFER_TOO_SMALL	0x0300	Most likely returned by EE_GetUserProfile() when the size of the supplied buffer is not large enough to hold the profile.
EDK_OUT_OF_RANGE	0x0301	One of the parameters supplied to the function is out of range.
EDK_INVALID_PARAMETER	0x0302	One of the parameters supplied to the function is invalid (e.g. pointers, zero size buffer)
EDK_PARAMETER_LOCKED	0x0303	The parameter value is currently locked by a running detection and cannot be modified at this time.
EDK_COG_INVALID_TRAINING_ACTION	0x0304	The specified action is not an allowed training action at this time.
EDK_COG_INVALID_TRAINING_CONTROL	0x0305	The specified control flag is not an allowed training control at this time.
EDK_COG_INVALID_ACTIVE_ACTION	0x0306	An undefined action bit has been set in the actions bit vector.
EDK_COG_EXCESS_MAX_ACTIONS	0x0307	The current action bit vector contains

		more than maximum number of concurrent actions.
EDK_EXP_NO_SIG_AVAILABLE	0x0308	A trained signature is not currently available for use – some actions may still require training data.
EDK_INVALID_USER_ID	0x0400	The user ID supplied to the function is invalid.
EDK_EMOENGINE_UNINITIALIZED	0x0500	EmoEngine™ needs to be initialized via calling IEE_EngineConnect() or
EDK_EMOENGINE_DISCONNECTED	0x0501	The connection with a remote instance of the EmoEngine made via EE_EngineRemoteConnect() has been lost
EDK_EMOENGINE_PROXY_ERROR	0x0502	The API was unable to establish a connection with a remote instance of the EmoEngine
EDK_STREAM_UNINITIALIZED	0x0503	There is no data stream with the userID
EDK_FILESTREAM_ERROR	0x0504	The stream is not file stream
EDK_STREAM_NOT_SUPPORTED	0x0505	The stream is not supported
EDK_FILE_ERROR	0x0506	The file error
EDK_NO_EVENT	0x0600	There are no new EmoEngine events at this time
EDK_GYRO_NOT_CALIBRATED	0x0700	The gyro is not calibrated. Please ask the user to stay still for at least 0.5 seconds
EDK_OPTIMIZATION_IS_ON	0x0800	Operation failure due to optimization
EDK_RESERVED1	0x0900	Reserved return value

EmoEngine Error Code	**Hex Value**	**Description**
EDK_PROFILE_CLOUD_EXISTED	0x1010	Profile created by EDK_SaveUserProfile() is existed in Emotiv Cloud.
EDK_UPLOAD_FAILED	0x1011	The file uploaded to cloud is failed
EDK_INVALID_CLOUD_USER_ID	0x1020	The cloud user ID supplied to the function is invalid.
EDK_INVALID_ENGINE_USER_ID	0x1021	The user ID supplied to the function is invalid
EDK_CLOUD_USER_ID_DONT_LOGIN	0x1022	The user ID supplied to the function dont login, call EDK_Login() first
EDK_EMOTIVCLOUD_UNINITIALIZED	0x1023	The Emotiv Cloud needs to be initialized via EDK_Connect()
EDK_FILE_EXISTS	0x2000	The file exist
EDK_HEADSET_NOT_AVAILABLE	0x2001	The headset is not available to work
EDK_HEADSET_IS_OFF	0x2002	The headset is off
EDK_SAVING_IS_RUNNING	0x2003	Other session of saving is running
EDK_DEVICE_CODE_ERROR	0x2004	Device ID code is error
EDK_LICENSE_ERROR	0x2010	The license is error.
EDK_LICENSE_EXPIRED	0x2011	The license expried
EDK_LICENSE_NOT_FOUND	0x2012	The license was not found
EDK_OVER_QUOTA	0x2013	The license is over quota
EDK_INVALID_DEBIT_ERROR	0x2014	Debit number is invalid

EmoEngine Error Code	**Hex Value**	**Description**
EDK_OVER_DEVICE_LIST	0x2015	Device list of the license is over
EDK_APP_QUOTA_EXCEEDED	0x2016	
EDK_APP_INVALID_DATE	0x2017	
EDK_LICENSE_DEVICE_LIMITED	0x2019	Application register device number is exceeded.
EDK_LICENSE_REGISTERED	0x2020	The license registered with the device
EDK_NO_ACTIVE_LICENSE	0x2021	No license is activated
EDK_LICENSE_NO_EEG	0x2022	The license is no EEG data ouput
EDK_UPDATE_LICENSE	0x2023	The license is updated
EDK_INVALID_DEBIT_NUMBER	0x2024	Session debit number is more then max of remaining session number
EDK_FILE_NOT_FOUND	0x2030	The file was not found
EDK_ACCESS_DENIED	0x2031	Access denied
EDK_NO_INTERNET_CONNECTION	0x2100	Could not connect to internet
EDK_AUTHENTICATION_ERROR	0x2101	An authentication error has occurred. Need to relogin
EDK_LOGIN_ERROR	0x2102	login error

Figure 24 Error code lists

Appendix 2 Emotiv EmoEngine™ Events

In order for an application to communicate with Emotiv EmoEngine, the program must regularly check for new EmoEngine events and handle them accordingly. Emotiv EmoEngine events are listed below

EmoEngine events	**Hex Value**	**Description**
IEE_UserAdded	0x0010	New user is registered with the EmoEngine
IEE_UserRemoved	0x0020	User is removed from the EmoEngine's user list
IEE_EmoStateUpdated	0x0040	New detection is available
IEE_ProfileEvent	0x0080	Notification from EmoEngine in response to a request to acquire profile of an user
IEE_MentalCommandEvent	0x0100	Event related to Cognitiv detection suite. Use the IEE_CognitivGetEventType function to retrieve the Cognitiv-specific event type
IEE_FacialExpressionEvent	0x0200	Event related to the Expressiv detection suite. Use the IEE_ExpressivGetEventType function to retrieve the Expressiv-specific event type.
IEE_InternalStateChanged	0x0400	Not generated for most applications. Used by Emotiv Control Panel to inform UI that a remotely connected application has modified the state of the embedded EmoEngine through the API.
IEE_EmulatorError	0x0001	EmoEngine internal error.

Figure 25 Emotiv Engine Event