

Emotiv

Software Development Kit

User Manual for Emotiv Example C++

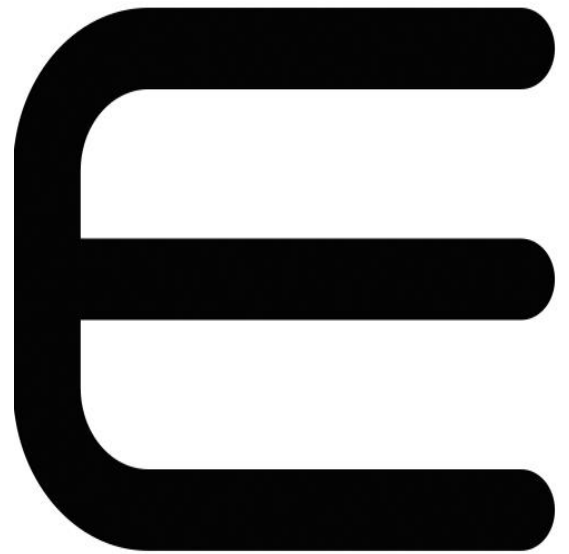


TABLE OF CONTENTS

| | |
|--|----|
| DIRECTORY OF FIGURES | 4 |
| DIRECTORY OF TABLES | 5 |
| DIRECTORY OF LISTINGS | 6 |
| 1. Introduction | 7 |
| 1.1 Glossary | 7 |
| 1.2 Trademarks | 8 |
| 2. Getting Started | 9 |
| 2.1 Hardware Components | 9 |
| 2.1.1 Charging the Neuroheadset Battery | 9 |
| 2.2 Emotiv libraries Installation | 10 |
| 2.2.1 Minimum Hardware and Software requirements | 10 |
| 2.2.2 Included Emotiv libraries software | 10 |
| 2.2.3 USB Receiver Installation | 10 |
| 2.2.4 Emotiv libraries Installation | 10 |
| 2.3 Start Menu Options | 13 |
| 3. Programming with the Emotiv libraries | 14 |
| 3.1 Overview | 14 |
| 3.2 Introduction to the Emotiv API and Emotiv EmoEngine™ | 14 |
| 3.3 Development Scenarios Supported by IEE_EngineRemoteConnect | 16 |
| 3.4 Examples non-EEG | 17 |
| 3.4.1 Example 1 – EmoStateLogger | 17 |
| 3.4.2 Example 2 – FacialExpressionDemo | 19 |
| 3.4.3 Example 3 – AverageBandPowers | 26 |
| 3.4.4 Example 4 – GyroData | 27 |
| 3.4.5 Example 5 – HeadsetInformationLogger | 28 |
| 3.4.6 Example 6 – Mental Commands Demo | 29 |
| 3.4.7 Example 7 – MotionDataLogger | 34 |
| 3.4.8 Example 8 – MultiDongleConnection | 35 |
| 3.4.9 Example 9 – SavingAndLoadingProfileCloud | 36 |
| 3.5 Examples EEG | 39 |
| 3.5.1 Example 1 – EEGLogger | 39 |
| 3.5.2 Example 2 – MultiDongleEEGLogger | 41 |
| 3.5.3 Example 3 – MultiChannelEEGLogger | 45 |
| 3.6 DotNetEmotivSDK Test | 47 |
| Appendix 1 EML Language Specification | 47 |
| A1.1 Introduction | 47 |
| A1.2 EML Example | 47 |
| A1.2.1 EML Header | 48 |

| | | |
|------------|--|----|
| A1.2.2 | EmoState Events in EML | 48 |
| Appendix 2 | Emotiv EmoEngine™ Error Codes | 54 |
| Appendix 3 | Emotiv EmoEngine™ Events | 56 |
| Appendix 4 | Redistributing Emotiv EmoEngine™ with your application | 57 |

DIRECTORY OF FIGURES

| | | |
|-----------|--|----|
| Figure 1 | Emotiv Xavier Setup _____ | 9 |
| Figure 2 | Emotiv libraries Setup wizard _____ | 11 |
| Figure 3 | Enter Key Order Number and Serial Key _____ | 12 |
| Figure 4 | Installation Complete dialog _____ | 13 |
| Figure 5 | Integrating the EmoEngine and Emotiv EPOC with a videogame _____ | 14 |
| Figure 6 | Using the API to communicate with the EmoEngine _____ | 15 |
| Figure 7 | Facial Expressions training command and event sequence _____ | 23 |
| Figure 8 | Mental Commands training _____ | 31 |
| Figure 9 | Normal and Triangle template shapes _____ | 52 |
| Figure 10 | Morphing a template _____ | 52 |
| Figure 11 | Morphed template _____ | 52 |

DIRECTORY OF TABLES

| | | |
|---------|---|----|
| Table 1 | BlueAvatar control syntax _____ | 20 |
| Table 2 | Time values in EML documents _____ | 49 |
| Table 3 | Detection groups in EML document _____ | 51 |
| Table 4 | Attributes for an event specification _____ | 53 |
| Table 5 | Emotiv EmoEngine™ Error Codes _____ | 55 |
| Table 6 | Emotiv EmoEngine™ Events _____ | 56 |

DIRECTORY OF LISTINGS

| | | |
|------------|---|----|
| Listing 1 | Connect to the EmoEngine | 18 |
| Listing 2 | Buffer creation and management | 18 |
| Listing 3 | Disconnecting from the EmoEngine | 19 |
| Listing 4 | Excerpt from Facial Expressions Demo code | 20 |
| Listing 5 | Extracting Facial Expressions event details | 23 |
| Listing 6 | Training "smile" and "neutral" in Facial Expressions Demo | 25 |
| Listing 7 | Log data of average band power | 26 |
| Listing 8 | Gyro Data | 28 |
| Listing 9 | Headset Information Logger | 29 |
| Listing 10 | Querying EmoState for Mental Commands detection results\ | 30 |
| Listing 11 | Extracting Mental Commands event details | 32 |
| Listing 12 | Training "push" and "neutral" with Mental CommandsDemo | 33 |
| Listing 13 | Get data of Motion | 35 |
| Listing 14 | Multi Dongle Connection | 36 |
| Listing 15 | Enter data for Emotiv Cloud | 37 |
| Listing 16 | Saving and Loading data from Cloud | 38 |
| Listing 17 | Access to EEG data | 39 |
| Listing 18 | Start Acquiring Data | 39 |
| Listing 19 | Acquiring Data | 40 |
| Listing 20 | Creat data1.csv and data2.csv for Multi Dongle EEGLogger | 41 |
| Listing 21 | Write data1.csv and data2.csv file | 45 |
| Listing 22 | Connect to the EmoComposer or EmoEngine | 46 |
| Listing 23 | Create and management buffer | 46 |
| Listing 24 | EML Document Example | 48 |
| Listing 25 | EML Header | 48 |
| Listing 26 | Sequence in EML document | 49 |
| Listing 27 | Configuring detections to automatically reset | 51 |

1. Introduction

This document is intended as a guide for Emotiv libraries developers. It describes different aspects of the Emotiv Xavier, including:

| | |
|--------------------------------|--|
| Getting Started | Basic information about installing the Emotiv Insight hardware and software. |
| Emotiv Xavier Tools | Usage guide for XavierEmoKey™ and XavierComposer™, tools that help you develop applications with the Emotiv Xavier |
| Emotiv API Introduction | Introduction to programming with the Emotiv API and an explanation of the code examples included with the SDK |

If you have any queries beyond the scope of this document, please contact the Emotiv SDK support team.

1.1 Glossary

| | |
|--------------------------|---|
| Performance Metrics | The detection suite that deciphers a user's emotional state. |
| Insight SDK Neuroheadset | The headset worn by the user, which interprets brain signals and sends the information to Emotiv Insight Driver. |
| Mental Command | The detection suite that recognizes a user's conscious thoughts. |
| Default Profile | A generic profile template that contains default settings for a new user. See Profile. |
| Detection | A high-level concept that refers to the proprietary algorithms running on the neuroheadset and in Emotiv Insight Driver which, working together, recognize a specific type of facial expression, emotion, or mental state. Detections are organized into four different suites: Mental Commands, Facial Expressions, Performance Metrics, Inertial Sensors. |
| EML | XavierComposer™ Markup Language – an XML-based syntax that can be interpreted by XavierComposer to playback predefined EmoState values. |
| Emotiv API | Emotiv Application Programming Interface: a library of functions, provided by Emotiv to application developers, which enables them to write software applications that work with Emotiv neuroheadsets and the Emotiv detection suites. |
| Emotiv EPOC™ | The neuroheadset that will be available with Emotiv's consumer product. |
| Emotiv Xavier | The Emotiv Software Development Kit: a toolset that allows development of applications and games to interact with Emotiv EmoEngine™ and Emotiv neuroheadsets. |
| XavierComposer™ | An Emotiv EmoEngine™ emulator designed to speed-up |

| | |
|-------------------|---|
| | the development of Emotiv-compatible software applications. |
| Emotiv EmoEngine™ | A logical abstraction exposed by the Emotiv API. EmoEngine communicates with the Emotiv neuroheadset, manages user-specific and application-specific settings, and translates the Emotiv detection results into an EmoState. |
| XavierEmoKey™ | Tool to translate EmoStates™ into signals that emulate traditional input devices (such as keyboard). |
| EmoScript™ | A text file containing EML, which can be interpreted by XavierComposer to automate the generation of predefined EmoStates. Also refers to the operational mode of XavierComposer in which this playback occurs. |
| EmoState™ | A data structure containing information about the current state of all activated Emotiv detections. This data structure is generated by Emotiv EmoEngine and reported to applications that use the Emotiv API. |
| Player | Synonym for User. |
| Profile | A user profile contains user-specific data created and used by the EmoEngine to assist in personalizing Emotiv detection results. When created with Emotiv Xavier, all users' profiles are saved to the profile.bin file in the Emotiv program files directory. |
| User | A person who is wearing a neuroheadset and interacting with Emotiv-enabled software. Each user should have a unique profile. |

1.2 Trademarks

The following are trademarks of Emotiv.

The absence of a product or service name or logo from this list does not constitute a waiver of Emotiv's trademark or other intellectual property rights concerning that name or logo.

Performance Metrics

Mental Commands

XavierComposer

XavierEmoKey™

EmoScript™

EmoState™

Emotiv EmoEngine™

Emotiv EPOC™

Emotiv SDK

Facial Expressions

2. Getting Started

2.1 Hardware Components

The Emotiv Xavier consists of one Insight SDK neuroheadsets, one USB wireless receivers, and an installation CD. The neuroheadsets capture users' brainwave (EEG) signals. After being converted to digital form, the brainwaves are processed, and the results are wirelessly transmitted to the USB receivers. A post-processing software component called Emotiv EmoEngine™ runs on the PC and exposes Emotiv detection results to applications via the Emotiv Application Programming Interface (Emotiv API).

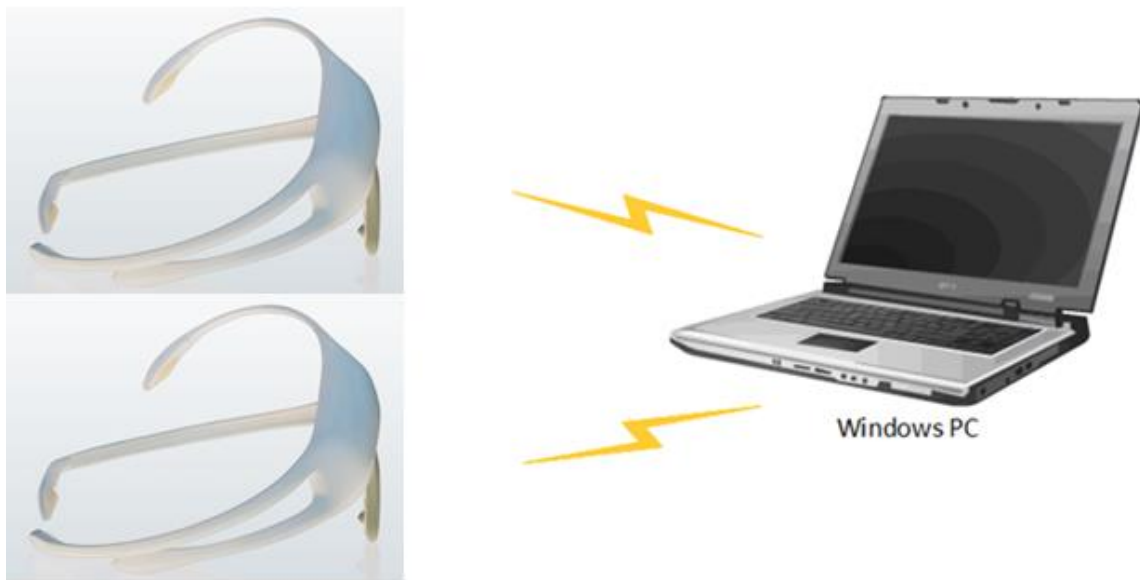


Figure 1 *Emotiv Xavier Setup*

For more detailed hardware setup and neuroheadset fitting instructions, please see the "Emotiv Xavier Hardware Setup.pdf" file shipped to SDK customers.

2.1.1 Charging the Neuroheadset Battery

The neuroheadset contains a built-in battery which is designed to run for approximately 12 hours when fully charged. To charge the neuroheadset battery, set the power switch to the "off" position, and plug the neuroheadset into the Emotiv battery charger using the mini-USB cable provided with the neuroheadset. Using the battery charger, a fully-drained battery can be recharged to 100% capacity in approximately 6 hours; charging for 30 minutes usually yields about a 10% increase in charge.

Alternatively, you may recharge the neuroheadset by connecting it directly to a USB port on your computer. Please note that this method takes the same amount of time to charge the battery.

The neuroheadset contains a status LED located next to the power switch at the back of the headband. When the power switch is set to the "on" position, the LED will illuminate

and appear blue if there is sufficient charge for correct operation. The LED will appear red during battery charging; when the battery is fully-charged, the LED will display green.

2.2 Emotiv libraries Installation

This section guides you through the process of installing the Emotiv Software Development Kit on a Windows PC.

2.2.1 Minimum Hardware and Software requirements

- 2.4 GHz Intel Pentium 4 processor (or equivalent).
- Microsoft Windows XP with Service Pack 2 or Microsoft Windows Vista.
- 1GB RAM.
- 50 MB available disk space.
- One or two unused USB 2.0 ports (depending on the number of neuroheadsets you wish to use simultaneously)

The program displays best when the DPI setting in Windows is 100%

2.2.2 Included Emotiv libraries software

Emotiv libraries developers will download the compressed file Emotiv standard libraries installer _v3.x.x.exe or Emotiv premium libraries installer _v3.x.x.exe, which contains both the Xavier libraries software and this User Manual.

Insight libraries developers will download the relevant Edition of the Insight SDK that has all software needed for Emotiv libraries installation. Log in to your account at www.emotiv.com and navigate to My Emotiv ~> Purchases. Your Insight SDK Edition should be available for download. Please also note the installation keys available from the KEY icon next to the DOWNLOAD button.

2.2.3 USB Receiver Installation

(This section is not relevant for SDK developers).

Plug the provided Emotiv USB receiver(s) into an unused USB port on your computer. Each receiver should be recognized and installed automatically by your computer as a USB Human Interface Device. The receivers follow the USB Human Interface Device standard so no additional hardware drivers are required to be installed. Please wait for a moment until Windows indicates that the new hardware is installed and ready to use.

2.2.4 Emotiv libraries Installation

This section explains the steps involved in installing the Emotiv libraries software. If an older version of the Emotiv libraries is present in your computer, we recommend that you uninstall it before proceeding.

Step 1 Using Windows Explorer, access the Emotiv standard libraries installer _v3.x.x.exe downloaded from the website.

Step 2 Run the Emotiv standard libraries installer _v3.x.x.exe or Emotiv premium libraries installer _v2.x.x.exe file. A window will appear after a few seconds.

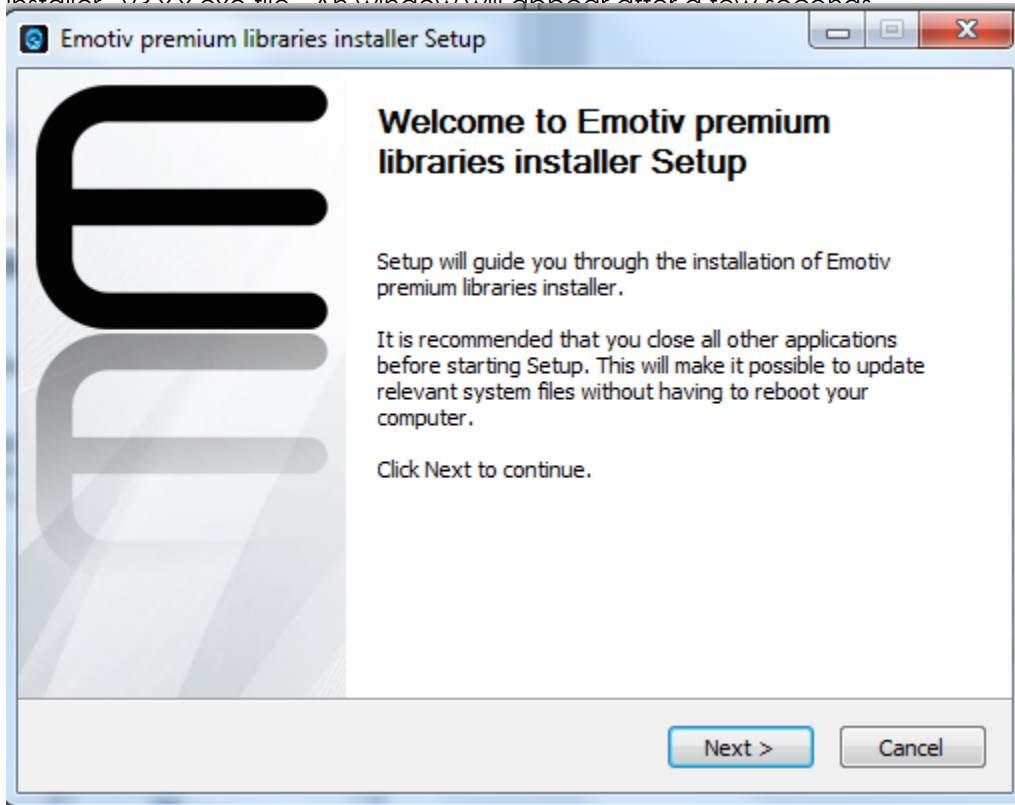
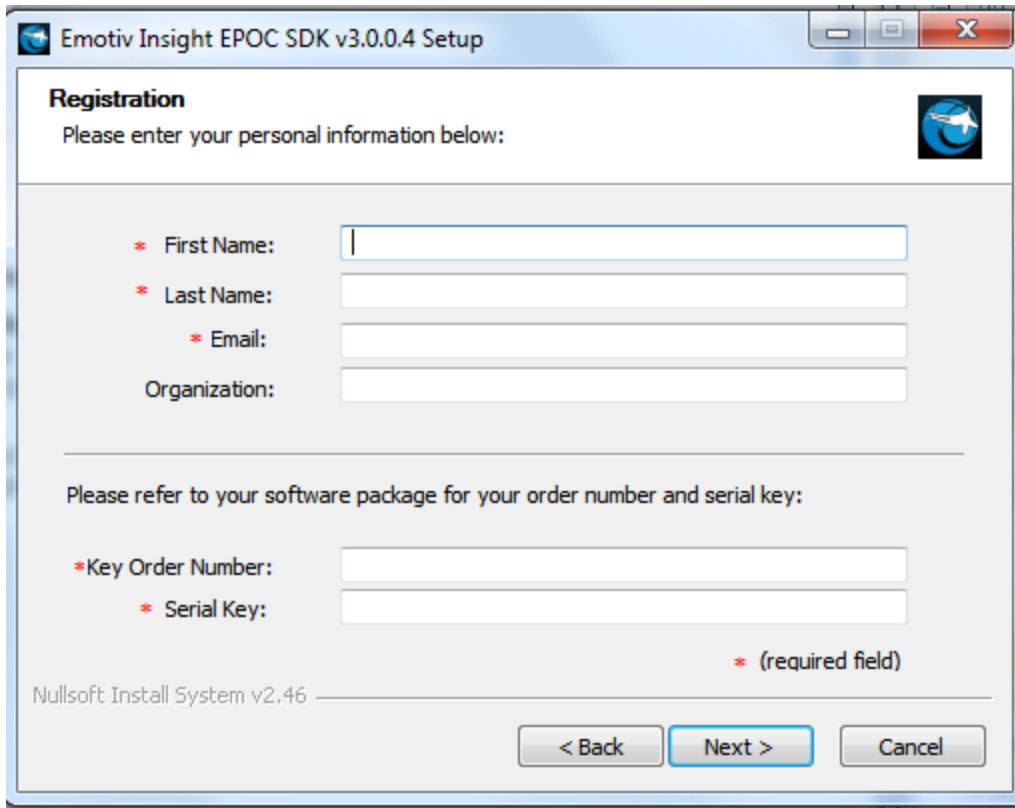


Figure 2 *Emotiv libraries Setup wizard*

Step 3 Click Next to start the installation process. You will be asked to enter First Name, Last Name, Email, Key Order Number and Serial Key. These numbers are available from the KEY icon next to the DOWNLOAD button at My Emotiv => Purchase. Enter these numbers and click Next.

Note: when you enter the correct Key Order Number and Serial Key, a pop-up box will appear indicating the "Serial Key is valid". Click OK and proceed to the next step.



The image shows a Windows-style setup window titled "Emotiv Insight EPOC SDK v3.0.0.4 Setup". The window has a light blue header bar with standard minimize, maximize, and close buttons. The main content area is white and contains the following elements:

- Registration** section header.
- Instruction: "Please enter your personal information below:"
- Four input fields, each preceded by a red asterisk (*):
 - First Name: [text box]
 - Last Name: [text box]
 - Email: [text box]
 - Organization: [text box]
- A horizontal separator line.
- Instruction: "Please refer to your software package for your order number and serial key:"
- Two input fields, each preceded by a red asterisk (*):
 - Key Order Number: [text box]
 - Serial Key: [text box]
- A red asterisk (*) followed by the text "(required field)" located below the Serial Key field.
- At the bottom left, the text "Nullsoft Install System v2.46" is visible.
- At the bottom right, there are three buttons: "< Back" (disabled), "Next >" (highlighted in blue), and "Cancel" (disabled).

Figure 3 *Enter Key Order Number and Serial Key*

Click **Next** to start the installation process.

Step 4 If you haven't uninstalled and older version of the Emotiv libraries, you may be asked if you wish to uninstall the older copy before proceeding. Multiple copies of the Insight can coexist on the same machine but you must be careful not to "mix and match" components from multiple installations.

Step 5 After a few seconds, an **Installation Complete** dialog will appear.

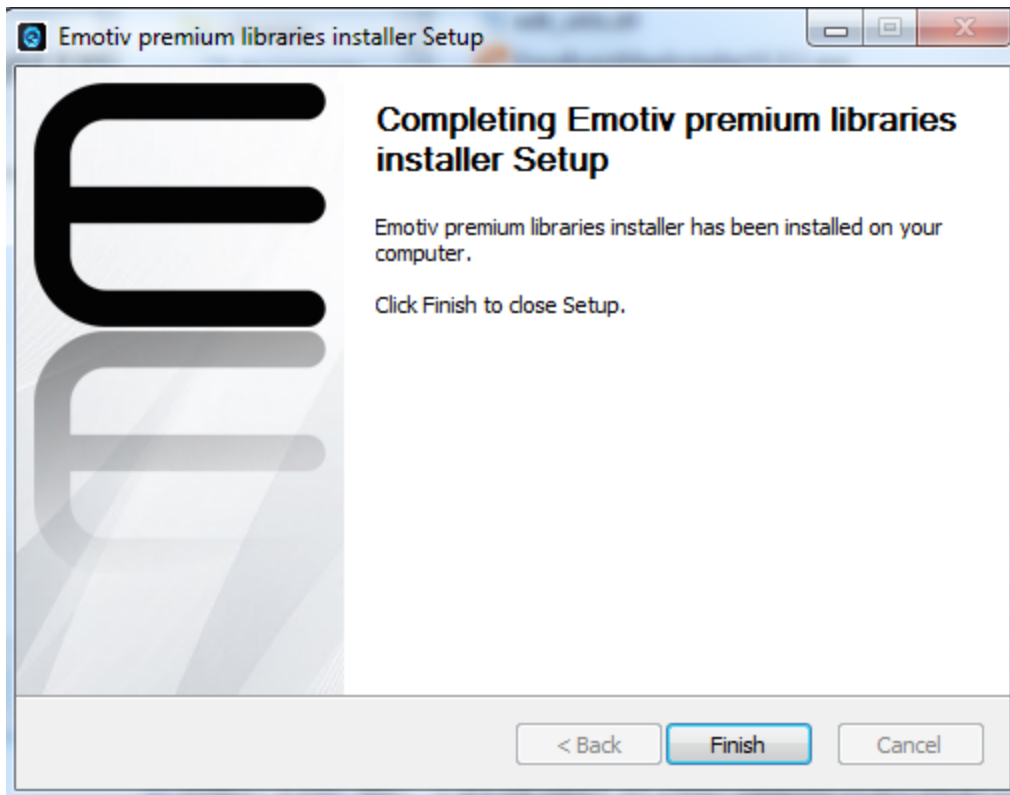


Figure 4 *Installation Complete dialog*

Step 6 Click **Finish** to complete the installation.

2.3 Start Menu Options

Once you have installed the Emotiv Insight, you will find the following in

Start > All Programs:

- 📁 Emotiv premium libraries installer v3.x.x
 - 📁 EDK
 - 📁 API References Emotiv API programmer's reference guide
 - 📁 Header files Headers for C#, C++, Java
 - 📁 X64 Edk.dll for x64
 - 📁 X86 Edk.dll for x86
 - 📁 Uninstall To uninstall the Emotiv Xavier

3. Programming with the Emotiv libraries

3.1 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 2010 (Visual Studio 2013 is also supported). They are stored at link <https://github.com/Emotiv/community-sdk> and are organized into a Microsoft Visual Studio 2010 solution, EmoTutorials.sln, which can be found in the \community-sdk\examples directory of your package you downloaded

With the Emotiv EEG examples you can get libraries in SDK

3.2 Introduction to the Emotiv API and Emotiv EmoEngine™

The Emotiv API is exposed as an ANSI C interface that is declared in 6 header files (ledk.h, IEmoStateDLL.h, ledkErrorCode.h, EmotivCloudClient.h, FacialExpressionDetection.h, MentalCommandDetection.h) and implemented in 2 Windows DLLs (edk.dll and edk_utils.dll). C or C++ applications that use the Emotiv API simply include ledk.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_."

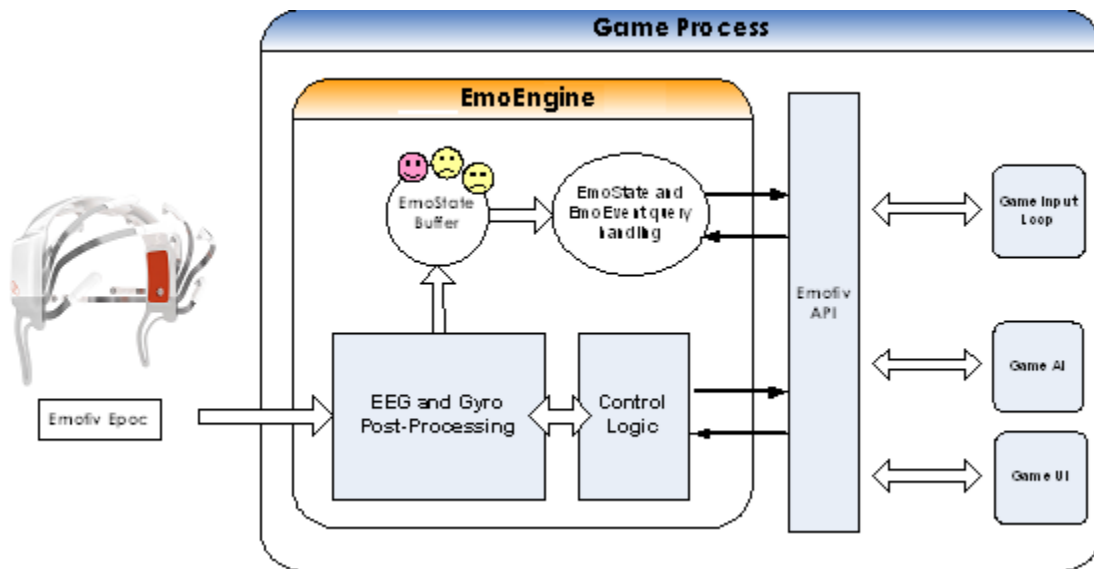


Figure 5 Integrating the EmoEngine and Emotiv EPOC with a videogame

An EmoState is an opaque data structure that contains the current state of the Emotiv detections, which, in turn, reflect the user's facial **expression** and Mental Commands state. EmoState data is retrieved by Emotiv API functions that are prefixed with "IES_." 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).

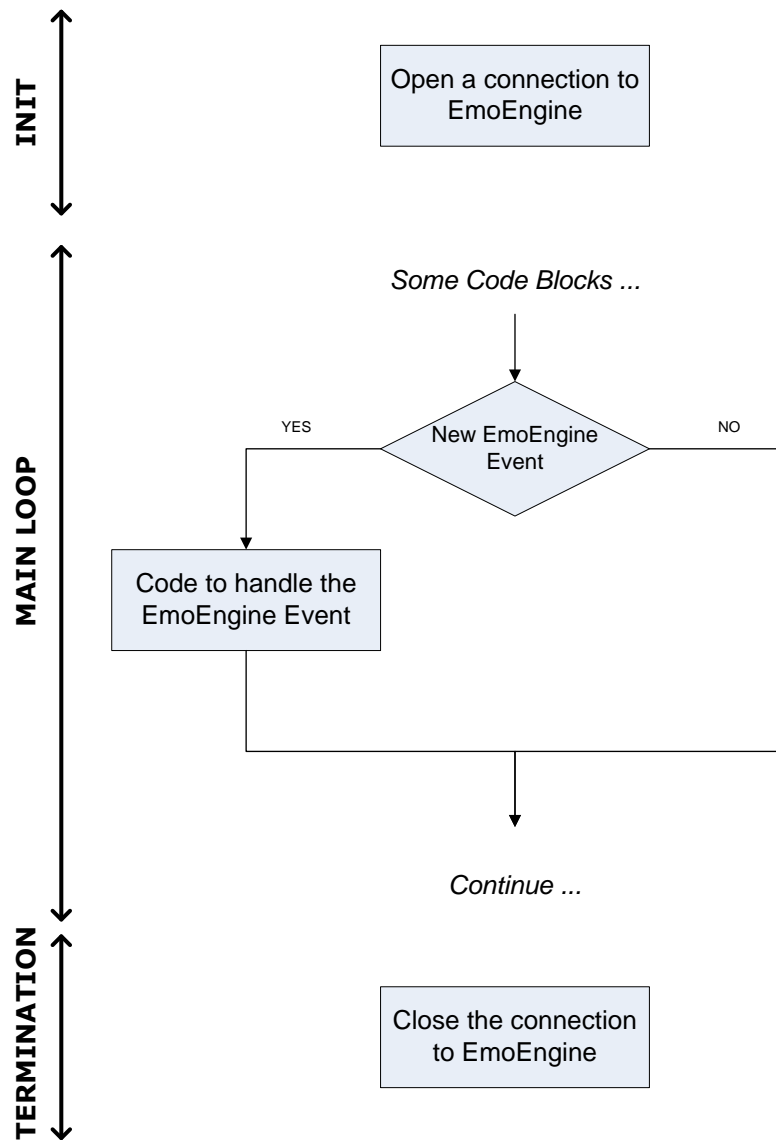


Figure 6 Using the API to communicate with the EmoEngine

Figure 6 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 XavierComposer or Emotiv Xavier. More details about using IEE_EngineRemoteConnect follow in Section 3.3.

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. 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 3**.

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 2**.

3.3 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 emulator. XavierComposer listens on port 1726 so an application that wishes to connect to an instance of XavierComposer 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 Xavier can act as a proxy for either the real, headset-integrated EmoEngine or XavierComposer. Insight SDK listens on port 3008 so an application that wishes to connect to Insight SDK must call `IEE_EngineRemoteConnect("127.0.0.1", 3008)`.
4. Emotiv Xavier 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 (if available) for a particular user.

```
/*!
```

```
- 0x50XX / 0x90XX - Insight Consumer
- 0x08XX / 0x09XX - Insight Premium
- 0x30XX / 0x70XX - EPOC+ Consumer
- 0x06XX / 0x07XX - EPOC+ Premium
- 0x1000 / 0x1E00 - EPOC Consumer
- 0x0565 - EPOC Premium
\param userId - user ID for query
\param pHwVersionOut - hardware version for the headset/dongle pair.
- Upper 2 bytes: headset version
- Lower 2 bytes: dongle version.
\return EDK_ERROR_CODE
- EDK_OK if successful
```

```
\sa IEmoStateDll.h, ledkErrorCode.h
```

we use 0x0565 for EPOC EEG, 0x1000 or 0x1E00 for non-EEG, 0x17B0 for Insight EEG, 0x0170 for Insight non-eeeg.

3.4 Examples non-EEG

3.4.1 Example 1 – EmoStateLogger

This example demonstrates the use of the core Emotiv API functions described in Sections 0 and 3.3. It logs all Emotiv detection results for the attached users after successfully establishing a connection to Emotiv EmoEngine™ or XavierComposer™.

```
// ... print some instructions...
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;
    }
}

```

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 XavierComposer 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 (...) {
    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 << std::endl;

            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 preallocated 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 XavierComposer . Start a new instance of EmoStateLogger and when prompted, select option 2 (**Connect to XavierComposer**). The EmoStates generated by XavierComposer 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 XavierComposer to generate any EmoStates. SDK users should only choose option 2 to connect to XavierComposer since option 1 (**Connect to EmoEngine**) assumes that the user will attach a neuroheadset to the computer.

3.4.2 Example 2 – FacialExpressionDemo

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 are 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: **I, S100**

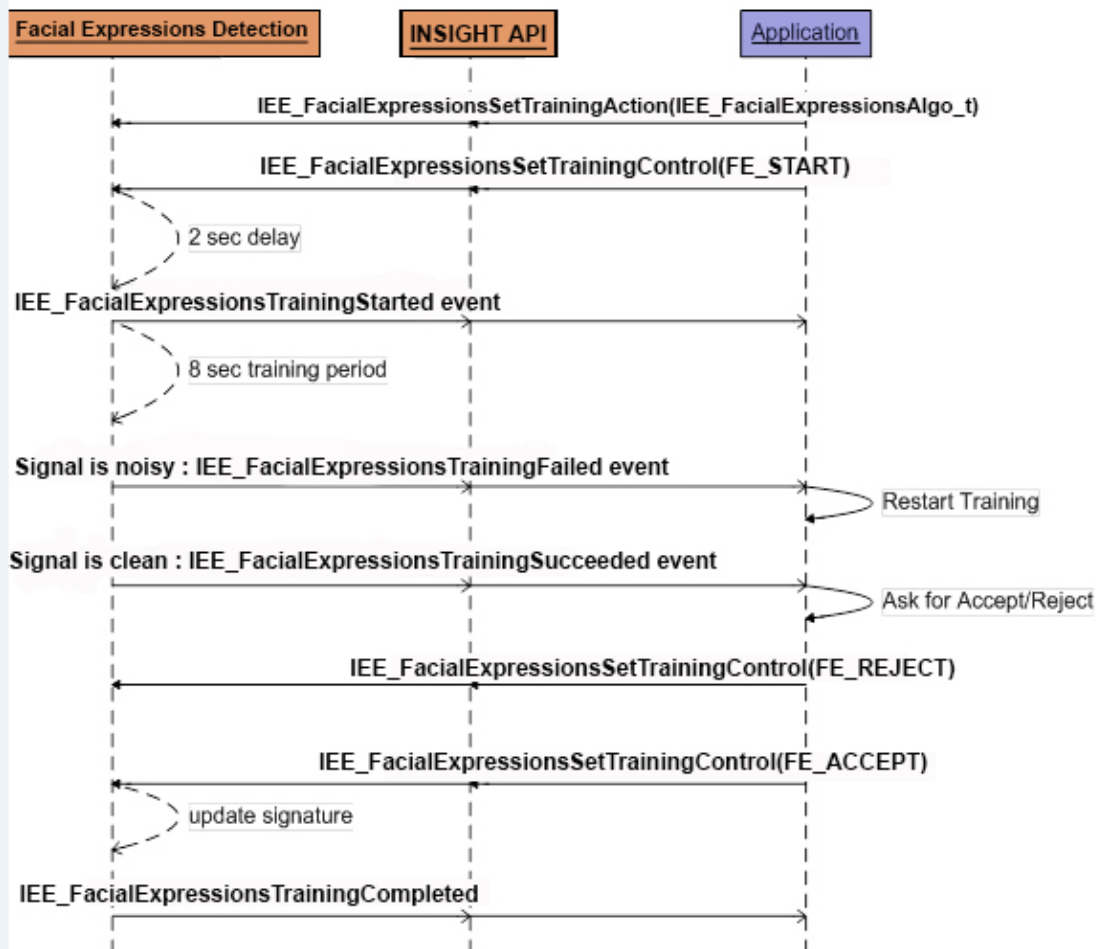
The prepared ASCII text is subsequently sent to the BlueAvatar via UDP socket. Facial Expressions Demo supports sending expression strings for multiple users. BlueAvatar should start listening to port 30000 for the first user. Whenever a subsequent Emotiv USB receiver is plugged-in, Facial Expressions 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") can not be trained.

The API functions that configure the Facial Expressions detections are prefixed with "IEE_FacialExpression." The **training_exp** command corresponds to the IEE_FacialExpressionSetTrainingAction() function. The **trained_sig** command corresponds to the IEE_FacialExpressionGetTrainedSignatureAvailable() 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 Xavier before attempting to use the Facial Expressions training API functions.



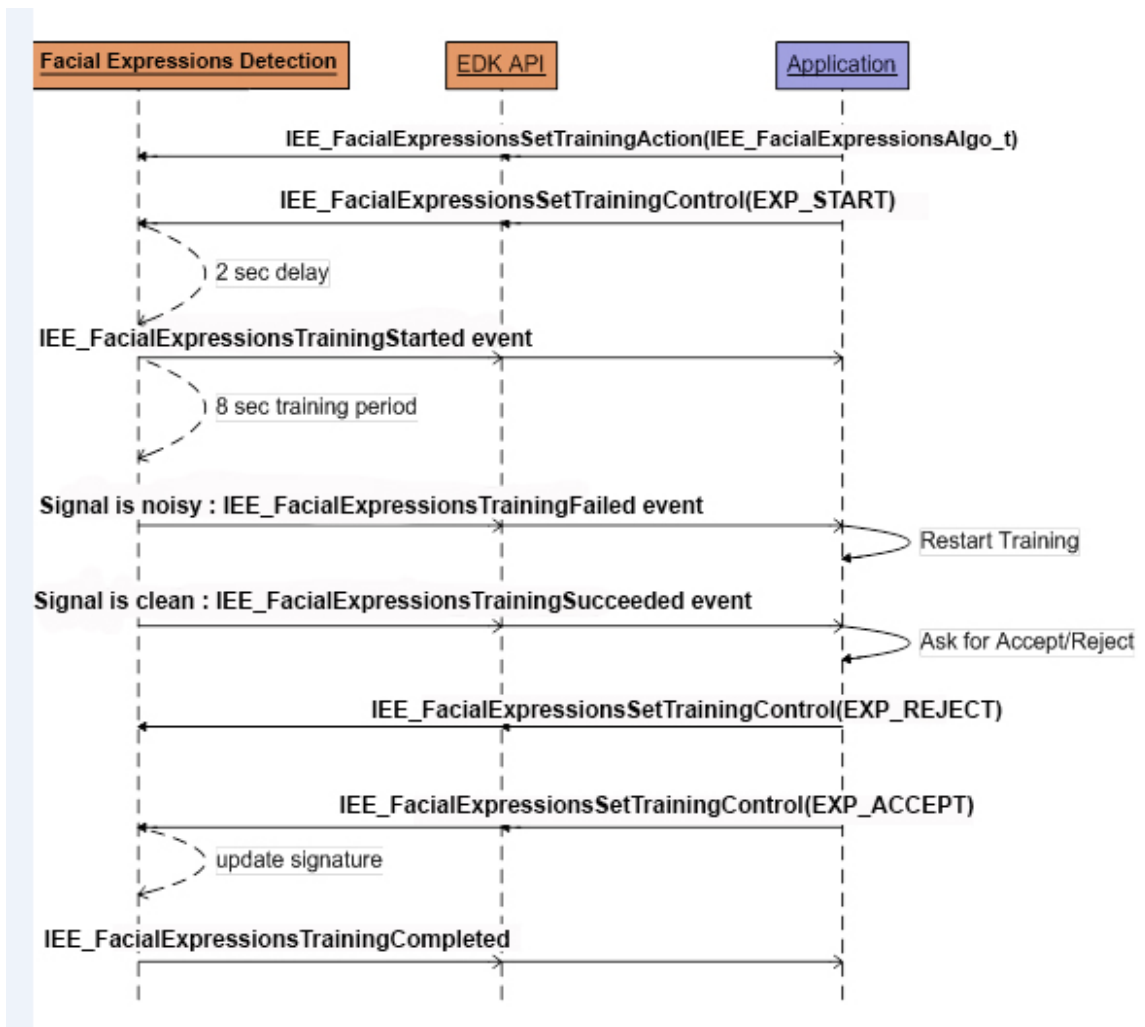


Figure 7 Facial Expressions training command and event sequence

The **Error! Reference source not found.** sequence diagram describes the process of training an Facial Expressions. The Facial Expressions -specific training events are declared as enumerated type `IEE_FacialExpressivEvent_t` in `EDK.h`. Note that this type differs from the `IEE_Event_t` type used by top-level EmoEngine Events.

```

EmoEngineEventHandle eEvent = 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 `EmoStateDLL.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 `EDK.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 EEG 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 EEG 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 EEG 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 `EXP_ACCEPT` or `EXP_REJECT`. If the training is rejected, then the application should wait until it receives the `IEE_FacialExpressionTrainingRejected` event before restarting the training process. If the training is accepted, EmoEngine™ will rebuild the user's trained Facial Expressions signature, and an `IEE_FacialExpressionTrainingCompleted` 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 Xavier and XavierComposer . In the Emotiv Xavier select **Connect→To XavierComposer** , 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 XavierComposer , 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*

3.4.3 Example 3 – AverageBandPowers

This examples demonstrates the API functions that can be used to get average band power for a specific channel from headset with 0.5 second step size and 2 seconds window size. Please note that this example is used for single connection.

A log file will be created with five columns : Theta, Alpha, Low_beta, High_beta, Gamma

```
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;
            engineUserID = 0;
            IEE_FFTSetWindowingType(engineUserID, IEE_HAMMING);

            std::cout << header << std::endl;
            ready = true;
        }

        if (ready)
        {
            double alpha, low_beta, high_beta, gamma, theta;
            alpha = low_beta = high_beta = gamma = theta = 0;

            for(int i=0 ; i< sizeof(channellist)/sizeof(channellist[0]) ; ++i)
            {
                int result = IEE_GetAverageBandPowers(engineUserID, channellist[i], &theta,
                &alpha, &low_beta, &high_beta, &gamma);
                if(result == EDK_OK){
                    ofs << theta << " ";
                    ofs << alpha << " ";
                    ofs << low_beta << " ";
                    ofs << high_beta << " ";
                    ofs << gamma << " ";
                    ofs << std::endl;

                    std::cout << theta << "," << alpha << "," << low_beta << ",";
                    std::cout << high_beta << "," << gamma << std::endl;
                }
            }
        }
    }
}
```

Listing 7 *Log data of average band power*

3.4.4 Example 4 – GyroData

Gyro data example allows built-in 2-axis gyroscope position. To get data of gyro data, you 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.

```
(...)
while(true)
{

    state = IEE_EngineGetNextEvent(hEvent);

    if (state == EDK_OK)
    {
        IEE_Event_t eventType = IEE_EmoEngineEventGetType(hEvent);
        IEE_EmoEngineEventGetUserId(hEvent, &userID);

        if (eventType == IEE_UserAdded) {
            std::cout << "User added" << std::endl << std::endl;
            userID = 0;
            ready = true;
        }

        if(!ready) continue;

        int gyroX = 0, gyroY = 0;
        int err = IEE_HeadsetGetGyroDelta(userID, &gyroX, &gyroY);

        if (err == EDK_OK){
            std::cout << std::endl;
            std::cout << "You can move your head now." << std::endl;

#ifdef _WIN32
                Sleep(1000);
#endif
            if __linux__ || __APPLE__
                usleep(10000);
            #endif

            break;
        }else if (err == EDK_GYRO_NOT_CALIBRATED){
            std::cout << "Gyro is not calibrated. Please stay still." << std::endl;
        }else if (err == EDK_CANNOT_ACQUIRE_DATA){
            std::cout << "Cannot acquire data" << std::endl;
        }else{
            std::cout << "No headset is connected" << std::endl;
        }
    }

#ifdef _WIN32
        Sleep(100);
#endif
}
```

```

#endif
#if __linux__ || __APPLE__
    usleep(10000);
#endif
}

#ifdef _WIN32
    globalElapsed = GetTickCount();
#endif
#if __linux__ || __APPLE__
    globalElapsed = ( unsigned long ) GetTickCount();
#endif

    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 8 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

3.4.5 Example 5 – HeadsetInformationLogger

This example allows user to get headset informations: Time, Wireless Strength, Battery Level, Contact Quality of AF3, T7, Pz, T8, AF4. All of these informations will be displayed on screen if you run example on Window or Linux.

This example only work on single connection

```

ofs.open(filePath.c_str());
ofs << "Time, Wireless Strength, Battery Level, AF3, T7, Pz, T8, AF4" << std::endl;

while (!_kbhit()) {

    state = IEE_EngineGetNextEvent(eEvent);
    if (state == EDK_OK) {

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

        switch (eventType)
        {
            case IEE_UserAdded:
                std::cout << "User added" << std::endl;

```

```

        readytocollect = true;
        break;
    case IEE_UserRemoved:
        std::cout << "User removed" << std::endl;
        readytocollect = false; //single connection
        break;
    case IEE_EmoStateUpdated:
        onStateChanged = true;
        IEE_EmoEngineEventGetEmoState(eEvent, eState);
        break;
    default:
        break;
    }
}

if (readytocollect && onStateChanged)
{
    onStateChanged = false;
    systemUpTime = IS_GetTimeFromStart(eState);
    wirelessStrength = IS_GetWirelessSignalStatus(eState);

    if (wirelessStrength != NO_SIG)
    {
        std::cout << "Time: " << systemUpTime << std::endl;
        IS_GetBatteryChargeLevel(eState, &batteryLevel,
&maxBatteryLevel);

        ofs << systemUpTime << ",";
        ofs << wirelessStrength << ",";
        ofs << batteryLevel << ",";
        ofs << IS_GetContactQuality(eState, IEE_CHAN_AF3) << ",";
        ofs << IS_GetContactQuality(eState, IEE_CHAN_T7) << ",";
        ofs << IS_GetContactQuality(eState, IEE_CHAN_Pz) << ",";
        ofs << IS_GetContactQuality(eState, IEE_CHAN_T8) << ",";
        ofs << IS_GetContactQuality(eState, IEE_CHAN_AF4) << ",";

        ofs << std::endl;
    }
}
}

```

Listing 9 Headset Information Logger

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.

3.4.6 Example 6 – 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 10.

```
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 10 *Querying EmoState for Mental Commands detection results*

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 Xavier 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 EEG 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 EEG signal.

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

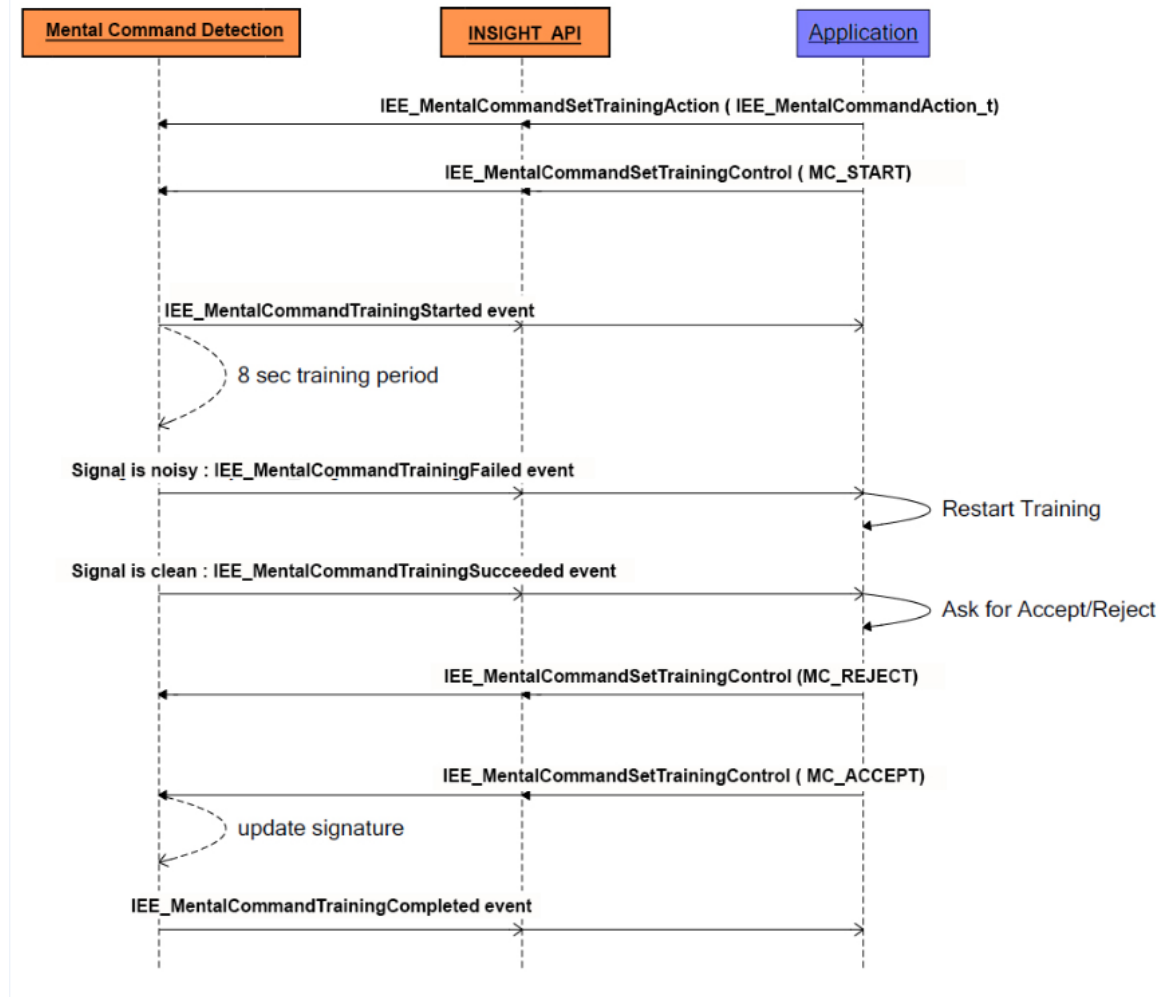


Figure 8 Mental Commands training

The **Error! Reference source not found.** 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 `EDK.h`. Note that this type differs from the `IEE_Event_t` type used by top-level EmoEngine Events. The code snippet in Listing 11 illustrates the procedure for extracting Mental Commands-specific event information from the EmoEngine event.

```

EmoEngineEventHandle eEvent = 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 11 *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 `EmoStateDLL.h`, the enumerated type `IEE_MentalCommandsAction_t` defines all the Mental Commands actions that are currently supported (`COG_PUSH`, `COG_LIFT`, etc.). If an action is not set before the start of training, `COG_NEUTRAL` will be used as the default.

`IEE_MentalCommandsSetTrainingControl()` can then be called with argument `COG_START` to start the training on the target action. In `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 `COG_START` command. The training update will begin after the `EmoEngine` sends the `IEE_MentalCommandsTrainingStarted` event. This delay will help to avoid training with undesirable EEG 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 EEG 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 EEG 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 `COG_ACCEPT` or `COG_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 Xavier` and the `XavierComposer`. In the `Emotiv Xavier` select **Connect→To XavierComposer** 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 XavierComposer the EmoCube responds accordingly.

Next, experiment with the training commands available in MentalCommandsDemo to better understand the Mental Commands training procedure described above. **Error! Reference source not found.** 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 12 *Training “push” and “neutral” with Mental CommandsDemo*

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);

```

3.4.7 Example 7 – MotionDataLogger

This example demonstrates how to extract live Motion data using the EmoEngine™ in C++. Data is read from headset and sent to an output file for later analysis. This example allows to show data of: Counter, GyroX, GyroY, GyroZ, AccX, AccY, AccZ, MagX, MagY, MagZ, Timestamp.

```

std::ofstream ofs(filename.c_str(),std::ios::trunc);
ofs << header << std::endl;

DataHandle hMotionData = IEE_MotionDataCreate();
IEE_MotionDataSetBufferSizeInSec(secs);

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

while (!_kbhit()) {

    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) {
            std::cout << "User added";
            readytocollect = true;
        }
    }

    if (readytocollect) {

        IEE_MotionDataUpdateHandle(0, hMotionData);

        unsigned int nSamplesTaken=0;
        IEE_MotionDataGetNumberOfSample(hMotionData, &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_MotionDataChannel_t) ;
                    i++) {

```

```

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

}

```

Listing 13 *Get data of Motion*

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.

3.4.8 Example 8 – MultiDongleConnection

This example demonstrates how to connect to two headsets at the same time.

This example also 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_FacialFacialExpressivGetSmileExtent(eState) <<endl;

        }

    }

}
// userremoved event
else if( eventType == IEE_UserRemoved )
{
    cout <<"user ID: "<<userID<<" have removed" <<

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

```

Listing 14 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

3.4.9 Example 9 – SavingAndLoadingProfileCloud

This example demonstrates how to save profile to Emotiv Cloud and load profile from Emotiv Cloud.

To run examples you need enter: your EmotivID, password and EmotivProfile to the sections corresponding as in Listing15.

```

std::string userName = "Your account name";
std::string password = "Your password";
std::string profileName = "EmotivProfile";

```

Listing 15 *Enter data for Emotiv Cloud*

Then you need to select Saving or Loading profile from Cloud

```
    if (IEE_EngineConnect() != EDK_OK) {
        std::cout << "Emotiv Driver start up failed.";
        return -1;
    }

    std::cout <<
    "=====
    << std::endl;
    std::cout << "Example to saving and loading profile from Emotiv Cloud. "
    << std::endl;
    std::cout <<
    "=====
    << std::endl;
    std::cout << "Press '1' to saving profile to Emotiv Cloud "
    << std::endl;
    std::cout << "Press '2' to loading profile from Emotiv Cloud "
    << std::endl;
    std::cout << ">> ";

    std::getline(std::cin, input, '\n');
    option = atoi(input.c_str());

    if(!IEC_Connect())
    {
        std::cout << "Cannot connect to Emotiv Cloud";
        return -2;
    }

    if(!IEC_Login(userName.c_str(), password.c_str()))
    {
        std::cout << "Your login attempt has failed. The username or password may be
incorrect";
#ifdef _WIN32
        _getch();
#endif
        return -3;
    }

    std::cout<<"Logged in as " << userName << std::endl;

    if (!IEC_GetUserDetail(&userCloudID))
        return -4;

    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);

        switch (option) {
            case 1:{
                int profileID = EC_GetProfileId(userCloudID, profileName.c_str());

                if (profileID >= 0) {
                    std::cout << "Profile with " << profileName << " is existed" << std::endl;
                    if (EC_UpdateUserProfile(userCloudID, engineUserID, profileID)) {
                        std::cout << "Updating finished";
                    }
                    else std::cout << "Updating failed";
                }
                else if (EC_SaveUserProfile(userCloudID, (int)engineUserID, profileName.c_str(),
TRAINING))
                {
                    std::cout << "Saving finished";
                }
                else std::cout << "Saving failed";

#ifdef _WIN32
                _getch();
#endif
                return -6;
            }
            case 2:{
                if (getNumberProfile > 0){
                    if (EC_LoadUserProfile(userCloudID,
(int)engineUserID, EC_ProfileIDAtIndex(userCloudID, 0)))
                        std::cout << "Loading finished";
                    else
                        std::cout << "Loading failed";
                }
            }
        }
    }
}

```

Listing 16 *Saving and Loading data from Cloud*

3.5 Examples EEG

3.5.1 Example 1 – EEGLogger

This example demonstrates how to extract live EEG data using the EmoEngine™ in C++. Data is read from the headset and sent to an output file for later analysis. **Please note that this example only works with the SDK versions Premium that allow raw EEG access.**

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 XavierComposer through a call to `IEE_EngineRemoteConnect()`. The EmoEngine event handlers and EmoState Buffer's are also created as before.

```
float secs = 1;
...
DataHandle hData = IEE_DataCreate();
IEE_DataSetBufferSizeInSec(secs);

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

Listing 17 Access to EEG data

Access to EEG 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 (!_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;
        }
    }
}
```

Listing 18 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 EEG for the user, storing it in the internal EmoEngine sample buffer. Note that the developer's application should access the EEG data at a rate that will ensure the sample buffer is not overrun.

```
    if (readytocollect)

...

        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(EpocChannelList) / sizeof(IEE_DataChannel_t);
i++) {

                    IEE_DataGet(hData, EpocChannelList[i], data, nSamplesTaken);
                    ofs << data[sampleIdx] << ",";
                }

                ofs << std::endl;
            }

            delete[] data;
        }
```

Listing 19 Acquiring Data

To initiate retrieval of the latest EEG buffered data, a call is made to IEE_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 IEE_DataUpdateHandle will be retrieved. Place a call to IEE_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_INTERPOLATED, IED_AF3, IED_F7, IED_F3, IED_FC5, IED_T7, IED_P7,
IED_O1, IED_O2, IED_P8, IED_T8, IED_FC6, IED_F4, IED_F8, IED_AF4, IED_RAW_CQ,
IED_GYROX, IED_GYROY, IED_MARKER, IED_TIMESTAMP
```

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);
```

3.5.2 Example 2 – MultiDongleEEGLogger

This example allows to get synchronized EEG data from two headsets. The data is only written to files as two headsets are in the good condition (without noise, full of battery, ...)

This example logs EEG data from two headset to data1.csv và data2.csv file in folder.

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

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

Listing 20 Creat data1.csv and data2.csv for Multi Dongle EEGLogger

Please note that this example only works with the SDK versions Premium that allow raw EEG access

Data1.csv or data2.csv file stores channels : IED_COUNTER, IED_INTERPOLATED, IED_RAW_CQ, IED_AF3, IED_T7, IED_Pz, IED_T8, IED_AF4, IED_TIMESTAMP, 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 samples_per_file = 1000;           // 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;

            if (numUsers > 2)
            {
                throw std::runtime_error("Too many users on demo!");
            }
        }
        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;
        }
        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;
        }

        // Don't overload */
        IsHeadset1On = false;
        IsHeadset2On = false;
    }
    }
}

```

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

Listing 21 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);
```

3.5.3 Example 3 – MultiChannelEEGLogger

This example is similar to the EEGLogger example, except using the IEE_DataGetMultiChannels function instead of IEE_DataGet(). It displays all data from all channels in channel list and logs to file after successfully establishing a connection to Emotiv EmoEngine™ or XavierComposer™.

```
// ... print some instructions...  
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;
}

```

Listing 22 Connect to the EmoComposer or 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 XavierComposer 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 (...) {
    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) {
            std::cout << "User added";
            IEE_DataAcquisitionEnable(userID,true);
            readytocollect = true;
        }
    }
}

```

Listing 23 Create and management buffer

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 preallocated EmoState buffer.

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

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

3.6 DotNetEmotivSDK Test

The Emotiv SDK comes with C# support. The wrapper is provided at \community-sdk\examples\C#\.

The test project at \community-sdk\examples\C#\DotNetEmotivSDK demonstrates how programmers can interface with the Emotiv SDK via the C# wrapper.

It's highly recommended that developers taking advantage of this test project read through other sections of these chapters. Concepts about the EmoEngine, the EmoEvents and EmoState are the same. DotNetEmotivSDK is merely a C# wrapper for the native C++ Emotiv SDK.

Appendix 1 EML Language Specification

A1.1 Introduction

XavierComposer™ is a hardware emulator for the Emotiv Software Development Kit. Using XavierComposer, game developers can emulate the behavior of Emotiv EmoEngine™ without needing to spend time in the real Emotiv EPOC™. XavierComposer operates in two modes, interactive and EmoScript playback.

In interactive mode, XavierComposer provides game developers with real time control over generating emulated detection events. XavierComposer also responds to a game's requests in real time. In EmoScript mode, game developers can pre-define these two-way interactions by preparing an XavierComposer Markup Language (EML) document. EML documents are XML documents that can be interpreted by XavierComposer. This section outlines the EML specification.

A1.2 EML Example

A typical EML document is shown in Listing 24 below:

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <!DOCTYPE EML>
03 <EML version="1.0" language="en_US">
04 <config>
05   <autoreset value="1" group="expressiv_eye" event="blink" />
06   <autoreset value="1" group="expressiv_eye" event="wink_left" />
07   <autoreset value="1" group="expressiv_eye" event="wink_right" />
08 </config>
09 <sequence>
10   <time value="0s15t">
11     <cognitiv event="push" value="0.85" />
12     <expressiv_upperface event="eyebrow_raised" value="0.85" />
13     <expressiv_lowerface event="clench" value="0.85" />
14     <expressiv_eye event="blink" value="1" />
15     <affectiv event="excitement_short_term" value="1" />
16     <affectiv event="excitement_long_term" value="0.6" />
17     <contact_quality value="G, G, G, G, G, G, F, F, G,
18       G, G, G, G, G, G, G, G, G" />
19   </time>
20   <time value="2s4t">
21     <cognitiv event="push" value="0" />
22     <expressiv_upperface event="eyebrow_raised" value="0.75" />
23     <expressiv_lowerface event="clench" value="0.5" />
```

```

24 <expressiv_eye event="blink" value="1" />
25 <affectiv event="excitement_short_term" value="0.7" />
26 <affectiv event="excitement_long_term" value="0.6" />
27 </time>
28 <time value="3s6t">
29 <cognitiv event="push" shape="normal" offset_left="0.4" offset_right="0.2"
30     scale_width="1.5" scale_height="0.8" />
31 <expressiv_upperface event="eyebrow_raised" value="0.85" />
32 <expressiv_lowerface event="clench" value="0.85" />
33 <expressiv_eye event="blink" value="1" repeat="1"
34     repeat_interval="0.5" repeat_num="15" />
35 <affectiv event="excitement_short_term" value="0.4" />
36 <affectiv event="excitement_long_term" value="0.5" />
37 </time>
38 </sequence>
39 </EML>

```

Listing 24 EML Document Example

Apart from standard headers (lines 1-3 and 39), an EML document consists of two sections:

- config: Section to configure global parameter for the XavierComposer behaviors.
- sequence: Section to define detection events as they would occur in a real Emotiv SDK.

A1.2.1 EML Header

Line 1-3 specifies the EML header. EML is a special implementation of a generic XML document which uses UTF-8 encoding and English US language. Line 2 is a normal XML comment to specify the document type and is optional.

```

01 <?xml version="1.0" encoding="utf-8"?>
02 <!DOCTYPE EML>
03 <EML version="1.0" language="en_US">

```

Listing 25 EML Header

A1.2.2 EmoState Events in EML

EmoState events are defined within the <sequence> element. In Listing 26, the <sequence> element is between line 9 and line 38:

```

09 <sequence>
10 <time value="0s15t">
11 <cognitiv event="push" value="0.85" />
12 <expressiv_upperface event="eyebrow_raised" value="0.85" />
13 <expressiv_lowerface event="clench" value="0.85" />
14 <expressiv_eye event="blink" value="1" />
15 <affectiv event="excitement_short_term" value="1" />
16 <affectiv event="excitement_long_term" value="0.6" />
17 <contact_quality value="G, G, G, G, F, F, P, F, G,
18     G, G, G, G, G, G, G, G, G" />
19 </time>
20 <time value="2s4t">
21 <cognitiv event="push" value="0" />

```



```

22 <expressiv_upperface event="eyebrow_raised" value ="0.75" />
23 <expressiv_lowerface event="clench" value ="0.5" />
24 <expressiv_eye event="blink" value="1" />
25 <affectiv event="excitement_short_term" value="0.7" />
26 <affectiv event="excitement_long_term" value="0.6" />
27 </time>
28 <time value="3s6t">
29 <cognitiv event="push" shape="normal" offset_left="0.4" offset_right="0.2"
30     scale_width="1.5" scale_height="0.8" />
31 <expressiv_upperface event="eyebrow_raised" value ="0.85" />
32 <expressiv_lowerface event="clench" value ="0.85" />
33 <expressiv_eye event="blink" value="1" repeat="1"
34     repeat_interval="0.5" repeat_num="15" />
35 <affectiv event="excitement_short_term" value="0.4" />
36 <affectiv event="excitement_long_term" value="0.5" />
37 </time>
38 </sequence>

```

Listing 26 Sequence in EML document

The <sequence> section consists of a series of discrete times at which there are events that will be sent from the XavierComposer to the game. These time events are ascending in time. Since each second is divided into 32 ticks (or frames), the time value in this example should be understood as follows:

| Time | Line Number | Description |
|-----------------|-------------|---|
| value = "0s15t" | 10 | This event is at 0 seconds and 15th frame |
| value = "2s4t" | 20 | This event is at 2 seconds and 4th frame |
| value = "3s6t" | 28 | This event is at 3 seconds and 6th frame |

Table 2 Time values in EML documents

At each time event, game developers can specify up to six different parameters, corresponding to the five distinct detection groups plus the current signal quality:

| Detection Group | Events | Notes |
|-----------------|---|-------|
| mental commands | push pull lift drop left right rotate_left rotate_right rotate_clockwise rotate_counter_clockwise rotate_forwards rotate_reverse | |

| | | |
|---------------------|---|--|
| | disappear | |
| expressiv_eye | blink wink_left wink_right look_left look_right | "value" attribute is treated as a boolean (0 or not 0) to determine whether to set the specified eye state. |
| expressiv_upperface | eyebrow_raised frown | |
| expressiv_lowerface | smile clench laugh smirk_left smirk_right | |
| affectiv | excitement_short_term excitement_long_term engagement_boredom | Notes: 1. The affectiv tag is a special case in that it is allowed to appear multiple times, in order to simulate output from all the Affectiv detections. 2. In order to simulate the behavior of the EmoEngine™, both short and long term values should be specified for excitement. |
| signal_quality | value | This tag has been deprecated. It has been replaced with the contact_quality tag. Expects "value" attribute to be formatted as 18 comma-separated floating point values between 0 and 1. The first two values must be the same. |
| contact_quality | value | Expects "value" attribute to be formatted as 18 comma-separated character codes that correspond to valid CQ constants: G = EEG_CQ_GOOD F = EEG_CQ_FAIR P = EEG_CQ_POOR VB = EEG_CQ_VERY_BAD NS = EEG_CQ_NO_SIGNAL |

| | | |
|--|--|--|
| | | <p>The first two values must be the same, and can only be set to G, VB, or NS, in order to most accurately simulate possible values produced by the Emotiv neuroheadset hardware.</p> <p>The order of the character codes is the same as the contents in the EE_InputChannels_enum declared in EmoStateDLL.h. Note that two of the channels, FP1 and FP2, do not currently exist on the SDK or EPOC neuroheadsets.</p> |
|--|--|--|

Table 3 Detection groups in EML document

Detection group names are created by grouping mutually exclusive events together. For example, only one of {blink, wink_left, wink_right, look_left, look_right} can happen at a given time, hence the grouping expressiv_eye.

Mental Commands detection group belongs to the Mental Commands Detection. Expressiv_eye, Expressiv_upperface, and Expressiv_lowerface detection groups belong to the Expressiv Detection Suite. Affectiv detection group belongs to the Affectiv Detection Suite.

In its simplest form, a detection definition parameter looks like: <cognitiv event="push" value="0.85" />, which is a discrete push action of the Cognitiv detection group with a value of 0.85. In EML, the maximum amplitude for any detection event is 1. By default, the detection event retains its value for this detection group until the game developer explicitly set it to a different value. However, game developers can also alter the reset behaviors as shown in the config section where the values for blink, wink_left, wink_right of the expressiv_eye detection group automatically reset themselves.

```

04 <config>
05   <autoreset value ="1" group="expressiv_eye" event="blink" />
06   <autoreset value ="1" group="expressiv_eye" event="wink_left" />
07   <autoreset value ="1" group="expressiv_eye" event="wink_right" />
08 </config>

```

Listing 27 Configuring detections to automatically reset

Instead of a discrete detection event as above, game developers can also define a series of detection events based on an event template function. An event template function generates a burst of discrete events according to the following parameters:

- shape: "normal" or "triangle"
- offset_left, offset_right, scale_width: A template has a 1 second width by default. These three parameters allow game developers to morph the template shape in the time domain.
- scale_height: A template, by default, has maximum amplitude of 1. This parameter allows game developers to morph the template's height.

Normal and Triangle shapes are shown below:

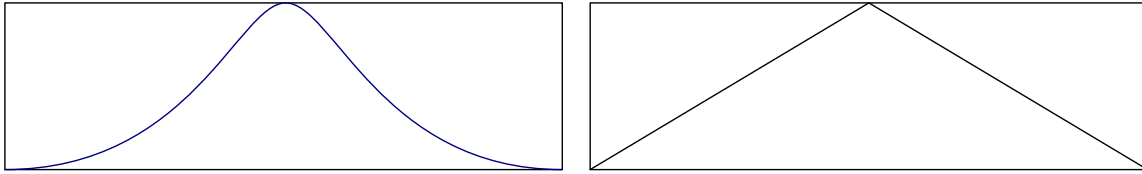


Figure 9 Normal and Triangle template shapes

An example of morphing template to specify detection event is:

```
29 <cognitiv event="push" shape="normal" offset_left="0.4" offset_right="0.2"
30     scale_width="1.5" scale_height="0.8" />
```

The above detection event can be illustrated as below:

First, start with a normal template with height = 1 and width = 1. Second, the template is adjusted by offset_left and offset_right. It now has a height of 1 and a width of $1 - 0.4 - 0.2 = 0.4$.

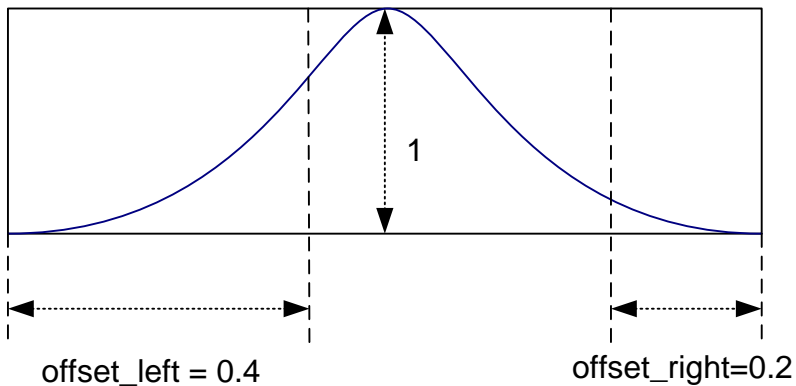


Figure 10 Morphing a template

Last, after height is scaled by scale_height and width is scaled by scale_width, the template becomes:

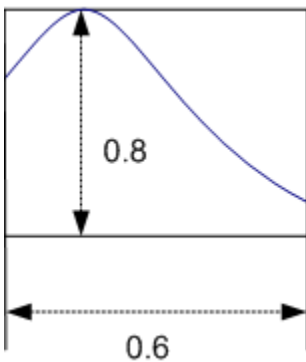


Figure 11 Morphed template

Full specifications of an event's attributes is shown below:

| Attribute | Description | Required |
|--------------------|--|----------|
| [detection_group] | One of six available detection groups as specified in Table 5. | Yes |
| event=[event_name] | Corresponding values of the | Yes |

| | | |
|-----------------------------|---|---|
| | [detection_group] as specified in Table 5. | |
| value=[value] | <p>A detection event can be interpreted as either a discrete event or a series of events whose values are determined by an event template function.</p> <p>The presence of the "value" attribute indicates that this is a discrete event.</p> | <p>Either "value" or "shape" attribute must be specified.</p> <p>If "value" is present, none of the event template attributes (shape, offset_left, offset_right, scale_width, scale_height) are allowed</p> |
| shape=[shape] | <p>The presence of the "shape" attribute indicates that this represents the starting point for a series of events generated according to an event template function.</p> <p>Allowed values are "normal" and "triangle".</p> | <p>Either "value" or "shape" attribute must be specified.</p> <p>If "shape" is present, then the "value" attribute is not allowed.</p> |
| offset_left=[offset_left] | <p>This attribute is a parameter of an event template function (see above for a detailed description of its meaning).</p> <p>offset_left+offset_right must be less than 1.</p> | <p>The "shape" attribute must also be specified.</p> <p>The "value" attribute can not be specified.</p> |
| offset_right=[offset_right] | <p>This attribute is a parameter of an event template function (see above for a detailed description of its meaning).</p> <p>offset_left+offset_right must be less than 1.</p> | <p>The "shape" attribute must also be specified.</p> <p>The "value" attribute can not be specified.</p> |
| scale_width=[scale_width] | <p>This attribute is a parameter of an event template function (see above for a detailed description of its meaning).</p> <p>Must be greater than 0.</p> | <p>The "shape" attribute must also be specified.</p> <p>The "value" attribute can not be specified.</p> |

Table 4 *Attributes for an event specification*

Appendix 2 Emotiv EmoEngine™ Error Codes

Every time you use a function provided by the API, the value returned indicates the EmoEngine™ status. Table 5 below shows possible EmoEngine error codes and their meanings. Unless the returned code is EDK_OK, there is an error. Explanations of these messages are in Table 5 below.

| 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 EPOCH™ 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. null 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 |

| EmoEngine Error Code | Hex Value | Description |
|-----------------------------|-----------|--|
| | | 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 IEE_EngineRemoteConnect() before calling any other APIs. |
| EDK_EMOENGINE_DISCONNECTED | 0x0501 | The connection with EmoEngine™ via IEE_EngineRemoteConnect() has been lost. |
| EDK_EMOENGINE_PROXY_ERROR | 0x0502 | Returned by IEE_EngineRemoteConnect() when the connection to the EmoEngine™ cannot be established. |
| EDK_NO_EVENT | 0x0600 | Returned by IEE_EngineGetNextEvent() when there is no pending event. |
| EDK_GYRO_NOT_CALIBRATED | 0x0700 | The gyroscope is not calibrated. Please ask the user to remain still for .5 seconds. |
| EDK_OPTIMIZATION_IS_ON | 0x0800 | Operation failed due to algorithm optimization settings. |

Table 5 *Emotiv EmoEngine™ Error Codes*

Appendix 3 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 in Table 6 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_CognitivEvent | 0x0100 | Event related to Cognitiv detection suite. Use the IEE_CognitivGetEventType function to retrieve the Cognitiv-specific event type. |
| IEE_ExpressivEvent | 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. |

Table 6 *Emotiv EmoEngine™ Events*

Appendix 4 Redistributing Emotiv EmoEngine™ with your application

An application constructed to use Emotiv EmoEngine™ requires that EDK.dll be installed on the end-user's computer. EDK.dll has been compiled with Microsoft Visual Studio 2005 (VC 8.0) SP1 and depends upon the shared C/C++ run-time libraries (CRT) that ship with this version of the compiler. The appropriate shared run-time libraries are installed on the application developer's machine by the Emotiv SDK™ Installer, but the developer is responsible for ensuring that the appropriate run-time libraries are installed on an end-user's computer by the developer's application installer before EDK.dll can be used on that machine.

If the application developer is using Visual Studio 2005 SP1+ to build her application then it is likely that no additional run-time libraries, beyond those already required by the application, need to be installed on the end-user's computer in order to support EDK.dll. Specifically, EDK.dll requires that Microsoft.VC80.CRT version 8.0.50727.762 or later be installed on the end-user's machine. Please see the following Microsoft documentation: [Redistributing Visual C++ files](#) and [Visual C++ Libraries as Shared Side-by-Side Assemblies](#) for more information about how to install the appropriate Microsoft shared run-time libraries or contact Emotiv's SDK support team for further assistance.

If the application is built using an older or newer major version of the Visual Studio compiler, such as Visual Studio 2003 or 2008, or another compiler altogether, then EDK.dll and the application will use different copies of the C/C++ run-time library (CRT). This will usually not cause a problem because EDK.dll doesn't rely on any shared static state with the application's instance of the CRT, but the application developer needs to be aware of some potentially subtle implications of using multiple instances of the CRT in the same process. Please refer to Microsoft's [C Run-Time Libraries \(CRT\)](#) documentation for more information on this subject. Depending on the particular compiler/run-time library mismatch involved, Emotiv may be able to provide a custom build of EDK.dll for developers who wish to use another compiler. Please contact the Emotiv SDK support team if you think you might require such a custom build.