AppConnector Example Documentation

J. Adam Wilson

2007-03-12

Contents

1 Introduction and Setup

1.1 AppConnectorExample

The AppConnectorExample program was designed as an example of how to use the AppConnector functionality for BCI2000, and as an example of a cross-platform application. The AppConnector functionality included in BCI2000 allows an external program to read and modify internal BCI2000 states (see the BCI2000 implementation guide and user manual for additional information about states). This program was written using Qt 4, which runs on Linux/Unix, Windows, and MacOS, and uses the platform-independent TCPStream class included with BCI2000 for TCP/UDP communication. ACE makes two network connections (sending and receiving) on two separate ports to a running BCI2000 session. The purposes of this documentation are:

1. Introduce the concept of cross-platform development using Qt 4
2. Provide an example program that can read BCI2000 states, locally or remotely
3. Show how to modify a state value, and send that state back to BCI2000

1.2 Qt4 Setup

Setting up Qt4 is largely dependent on your platform, and the distribution used (if on linux).

1.2.1 Windows

Installing Qt4 is reasonably straightforward on Windows. Go to http://www.trolltech.com/products/qt/downloads, and download the version that includes MinGW (it should say MinGW in the filename, as opposed to src). MinGW is a compiler for windows, and is needed because the open source version of Qt cannot be used with MS Visual Studio or the Borland Compiler. Download and setup Qt and MinGW. Next, you need to add the Qt/bin and MinGW/bin directories to your path. Do this by right-clicking on My Computer->Properties, go to Advanced, and click on Environment Variables. Under system variables, select Path, and add
`c:\Qt\4.2.3\bin;c:\MinGW\bin;`
to the FRONT of the existing path. Note that this does depend on where you actually installed qt and mingw, so adjust these directories accordingly.
One caveat to be aware of if you also have Cygwin installed on windows (if you do not, you can safely skip this) is that some of the files and programs can conflict if MinGW and Cygwin are present simultaneously. The quick and dirty solution is to start a Cygwin shell, and rename the `sh.exe` file, like:
`# mv /usr/bin/sh /usr/bin/shold`.
(*A better solution may be to move sh to shold, and then link to bash, since sh is recreated every time a bash shell starts; I have not tried this yet though.*

To work in the qt4 environment, go to the Start Menu->Qt 4 (Trolltech...), and select the Qt 4 Environment program. This opens a command window with all of the path variables set correctly, and will help reduce conflicts with Cygwin and possibly other compilers on the system. To use the mingw make program, you must type `mingw32-make`, and not just make.

1.2.2 Linux

This will depend on your distribution. Under gentoo (which I use), simply type `# emerge -av qt`. The newest version will be downloaded, compiled, and installed. Under debian, kubuntu, or ubuntu, type `sudo apt-get install libqt4-dev libqt4-gui libqt4-core libqt4-debug`. This may be wrong, since I do not have debian or kubuntu installed, but it should be in the ballpark, and if you are using these distros you should know what you are doing anyway :-). Also, make sure that you have gcc installed.

1.2.3 OS X

Note: I do not have OS X, nor have I ever used it. The Trolltech site at http://www.trolltech.com/products/qt/downloads has a section for Macs, so download and install the program according to those instructions, and hope for the best...and stop putting the letter 'i' in front of everything, it's annoying.

1.3 Development Environment

Unfortunately, the free version of Qt4 does not integrate with MS Visual Studio or Borland Developer Studio. However, there are many free IDE's available for C++. On Windows, DevC++ is a good choice, and has syntax highlighting, code completion, and integrates well with MinGW. On Linux, KDevelop and/or Kate (KDE Advanced Text Editor) are great choices. Mac people, again, you are on your own here.

Qt on linux also comes with Qt Designer, which allows for the quick development of forms and user interfaces. However, it does not include code generation; it saves an xml file describing the form, which uses an intermediate compiler called uic (user interface compiler) to generate the c++ code, which is THEN integrated with your code. I personally think it is easer to just write the GUI code by hand, and in this example this is what is done.

Another open source IDE available for Qt development is called QDevelop. This looks promising, since it combines the Qt Designer RAD interface with integrated code generation, and includes a code editor. I have not had a chance to check it out though.

2 Qt4 Development

2.1 Signals/Slots

Making the move to Qt4 can be a little confusing at first if you are used to how Borland or Microsoft handle things in C++, or Visual Basic/C#. In Borland C++ and MSVB or C#, GUI events directly trigger a callback function, which executes code and returns (i.e. you press a button, the code in the button callback is executed). Qt4 handles things a little differently. Qt4 implements what is called a SIGNAL/SLOT system, in which an object (button, list, any GUI element) emits a SIGNAL when a particular event occurs. This SIGNAL is connected to the SLOT of another object. All Qt4 objects have a number of predefined SIGNALS and SLOTS for the standard actions associated with such an object; for example, the most-commonly used SIGNAL for a button object is probably clicked(). To connect a SIGNAL and SLOT, use the `connect` command. Here is an example using two Qt objects:

```
QPushButton *button(‘‘Quit’’);
QApplication *app;
connect(button, SIGNAL(clicked()), app, SLOT(quit()));
```

In this example, a button pointer with the caption “Quit” is created, and the Qt4 Application is created. The button clicked() event is then tied to the quit() SLOT function for that app. So when the button is clicked, it calls the quit() function, which closes the window and application. As a result, many of the common tasks for an object are already defined, and you do not need to code them. An entire simple application might look like:

```
#include <QApplication>
#include <QPushbutton>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QPushButton button(‘‘Quit’’);
   connect(&button, SIGNAL(clicked()), &app, SLOT(quit()));
   button.show();
   return app.exec();
}
```

2.2 Extending Widgets

Even though there are many useful predefined SLOTS, you will eventually need to create your own functions to handle events. This is done through subclassing. In Qt (and C++), it is important to divide components into their own

classes. This includes the GUI. Therefore, when creating a form, you will want to create a class that inherits from a Qt object, most likely either a QWidget or QMainApplication. (QWidget is basically a completely blank form, while QMainApplication comes with a menu bar, a status bar, and other common features of an application). This class will then handle creating and setting up all of hte the GUI components. Importantly, inheriting from a QWidget (or QObject if it is not necessarily a user interface) allows us to create our own SIGNALS and SLOTS. So, here is a simple example class that inherits from QWidget, contains a button, and creates a custom SLOT.

```
//mainUI.h-------------

//include the components necessary to create a Qt GUI
#include <QtGui>

class QPushButton;

//create our form's class, inheriting from the QWidget class
class mainUI : public QWidget
{
    //the Q_OBJECT macro is necessary for creating SIGNALS/SLOTS,
    // and must be declared at the start of the class
    Q_OBJECT

    //SLOTS are created using the private slots:
    //declaration in the class definition
    //slot functions are declared and implemented
    //just as normal class methods
    private slots:
        void customButtonCallback();

    private:
        //our button
        QPushButton *button;
        //this layout arranges the object vertically, see Qt docs for more
        QVBoxLayout *mainLayout;

    public:
        //our class constructor
        mainUI();
};

//mainUI.cpp------------------------------
//here is our implementation
mainUI::main()
```

```
{
    button = new QPushButton("Push Me");
    mainLayout = new QVBoxLayout;

    //connect our button signal to our custom slot
    //since it is defined within this object, use the this keyword
    // for the SLOT target
    connect(button, SIGNAL(clicked()), this, SLOT(customButtonCallback()));

    //setup the layout
    mainLayout->addWidget(button);
    setLayout(mainLayout);
}

void mainUI::customButtonCallback()
{
    //this function is called whenever the button is pressed
    //do what you want here
}

//main.cpp---------------------------------
//this file starts the app
#include <QApplication>
#include "mainUI.h"

int main(int argc, char *argv[])
{
    //create the QT application
    QApplication app(argc, argv);

    //create our custom widget class
    mainUI mainWin;
    //show it
    mainWin.show();
    //execute the program
    return app.exec();
}
```

3 AppConnector Development

To start creating an AppConnector program, some BCI2000 source files must be included: the TCPStream.cpp and TCPStream.h files. This program includes these file in the src directory, but your program must include them manually, either by adding them to your includes and depends path (see the Qt4 qmake documentation), or by copying them to a local folder. These files contain the class definitions for the UDP socket and stream objects, which are used to read and write states with BCI2000. Create these objects as follows:

```
#include "TCPStream.h"

...

receiving_udpsocket recSocket;
tcpstream recConnection;
sending_udpsocket sendSocket;
tcpstream sendConnection;
```

To setup the connection, you first open the socket connection to an IP address and port using a string:

```
recSocket.open(“localhost:20320”);
```

The stream is initialized using the socket object created:

```
recConnection.open(recSocket);
```

Check if the connection is open:

```
if (!recConnection.is_open())
    //some error message...
```

The sending connection is setup the same way:

```
sendSocket.open("localhost:20321");
sendConnection.open(sendSocket);
if (!sendConnection.is_open())
    //error...
```

Note that you will not likely want to hard-code the IP and port. Providing a text box for these options in your program and using the text field to set the address is more functional.
To setup BCI2000 to work with your program, in the Config menu, go to the

	BCI2000	AppConnector
IP Address	192.168.0.100	192.168.0.101
ConnectorOutputAddress	192.168.0.101:20230	192.168.0.100:20231
ConnectorInputAddress	192.168.0.100:20231	192.168.0.101:20230

Table 1: Connections

ConnectorFilter tab. Under ConnectorOutputAddress, use the IP and Port used in the **Receving** address in the external program. So, if the receiving port and IP in the AppConnector program are `localhost:20320`, then the ConnectorOutputAddress in BCI2000 should be `localhost:20320`. Similarly, if the sending port and ip in the AppConnector program are `localhost:20321`, then the ConnectorInputAddress in the BCI2000 config should be `localhost:20321`. Since a network protocol is used for communication, the AppConnector program does not have to be on the same machine that is running BCI2000. For example, your desktop could be running BCI2000 on Windows, and your laptop running linux can run the AppConnector and connect to BCI2000. To do this, use the actual machine IP addresses for each computer. For example, if the PC with BCI2000 has an IP of 192.168.0.100 and the laptop has an IP of 192.168.0.101, then on the BCI2000 PC, the ConnectorOutputAddress should be set to 192.168.0.101:20320, the ConnectorInputAddress to 192.168.0.100:20321; on the laptop, the sending address should be 192.168.0.100:20321, and the receiving address should be 192.168.0.101:20320 (Table 1)

Once your connection is setup, using the tcpstream objects is simple. A function that reads data from the stream and stores the states and associated values in a map might look like this:

```
bool readData(map<string, float> &map, tcpstream &recConnection)
{
   int count = 0;

   //while there is data available in the buffer, read it
   while (recConnection.rdbuf()->in_avail())
   {
       string name;
       float value;

       //read the data using the stream format (i.e. >> operator)
       recConnection >> name >> value;
       recConnection.ignore();

       //check if there is some kind of error, and ignore it
       if (!recConnection)
           recConnection.ignore();
```

```
        map[name] = value;
        count++;
    }

    //if we read and recorded data, return true
    if (count > 0)
        return true;
    else
        return false;
}
```

After this function is called, the map contains the names and values of all of the states. To access the state value, use its name as the input to the [] operator, like:
`float value = map[“Feedback”];`

Writing data is even simpler using the stream format. Here is a short example:

```
string name = "Feedback";
short value = 0;
sendConnection << name << ’ ’ << value << endl;
```

That's it! BCI2000 handles all of the checks for whether the state exists and if it was a valid value.
To finish, here is a simple example that reads a state value, modifies it, and sends it back to BCI2000:

```
#include <map>
#include <string>
#include "TCPStream.h"

using namespace std;
int main() {
   receiving_udpsocket recSocket;
   tcpstream recConnection;
   sending_udpsocket sendSocket;
   tcpstream sendConnection;

   //open the sockets and streams (error checking as been removed)
   sendSocket.open("localhost:20321");
   sendConnection.open(sendSocket);

   recSocket.open("localhost:20320");
   recConnection.open(sendSocket);
```

```
    map<string, float> states;
    while (true) {
        //try to read the states into the map
        if (readData(&states, &recConnection))  {
            if (states["Feedback"] == 0)
                continue;

            //exit if BCI2000 has stopped
            if (states["Running"] == 0)
                return 0;

            float value = states["CursorPosX"];
            //change the value by 100, then write it to the output
            value += 100;
            sendConnection << "CursorPosX " << value << endl;

            //note that this actually requires a slight
            //modification of the d2box program in the task.cpp file
            //the ReadStates() function must be updated to include
            //x_pos = State("CursorPosX");
            //y_pos = State("CursorPosY");
        }
    }
}
```

4 AppConnector Example

This section focuses on the AppConnector example program, including how to build it and what it does.

4.1 Building

Instructions for building the project for Linux and Windows follow.

4.1.1 Linux

Qt4 uses a function called qmake to automatically generate the Makefile for the project. It reads from a file with the .pro extension, and creates a Makefile based on the contents of the .pro file (see the Qt documentation for qmake for additional info). To create the Makefile, just type `qmake` in the AppConnectorExample folder (not the AppConnectorExample/src folder). Once this done, type `make`, and the project will be built. The executable is placed in the `AppConnectorExampole/bin` directory, and is called AppConnectorExample. To run it, cd to the bin directory, and type `./AppConnectorExample`. It should start and run.

4.1.2 Windows

To enter the Qt4 development environment, go to the Start Menu, select Qt4, and open the Qt4 Development command prompt. cd to the directory containing the AppConnectorExample. As in linux, the `qmake` function generates the approprate Makefile for the project, so do this now. When this is complete, you use the mingw make command to build the project. On Windows, you have the option of building a debug or release version. For now, do the release, because getting the debuggable version of Qt4 working can be difficult. To make the project, type `mingw32-make debug` in the AppConnectorExample folder. The executable is placed in the release/bin folder.

4.1.3 OS X

Someone who has tried Qt4 on a Max will have to let me know how well this works. I assume it is very similar to Linux/Unix.

4.2 Running

After the project is built, run the program, and setup the connections as in Section 3 and Table 1. Once BCI2000 is running locally or remotely, press the Connect button to connect to BCI2000 on the addresses specified. If it works, the status box under the button will read "Receive = true Send = true" If there is an error, one or both will read false.
When BCI2000 is in a suspended state, it is not writing any data to the UDP socket, so the program does not read them. When you press Start, BCI2000

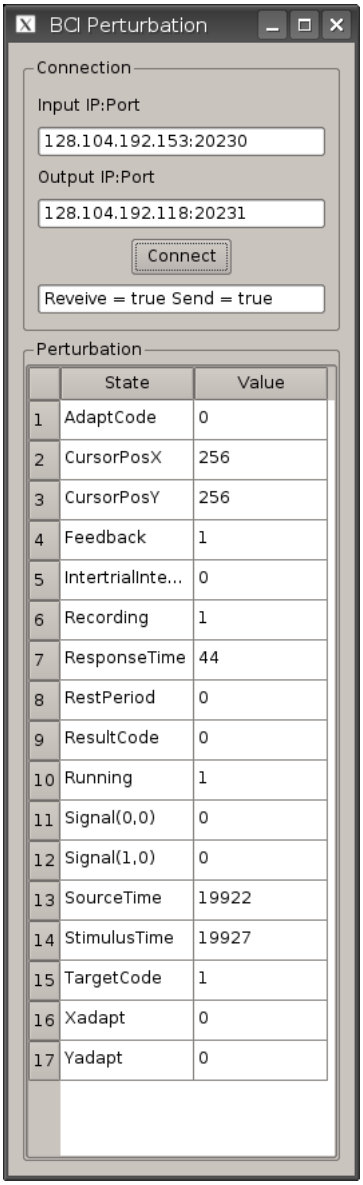

Figure 1: AppConnectorExample

begins writing the states to the socket, and the AppConnectorExample program will start reading them. The state names and values are displayed in the table at the bottom, and are updated every 100 ms.
To change a value, double-click the value column of the state you want to modify. A dialog box appears asking for the new value. Enter the value, and press Ok. The new value is written to BCI2000. For example, to stop the run, you can set the state Running to 0.