VRPN - Android

User & Developer Manuals

Eric Boren

Duncan Lewis

Ted Driggs

Kristen Janick

USER MANUAL

As a User of the Android Application, you will be using the phone as a device to control your interaction with a Virtual World. Once the application is launched, the screen will have a varying number of controls (buttons, sliders, and trackers) that map to functionalities in the Virtual World you are connected to. Each version of our application will always have an IP Address and Port Number found at the top of the screen. This is the connection that the Virtual World will use to connect to the phone. The application may also have any of the additional features which are outlined below. The functionality of each feature (ie. Buttons, trackers, etc.) will differ depending on the virtual world application they are interacting with. Activation of all features are as described below.

- **Buttons** – Simply tap the screen where the button is located to activate the button. There are two buttons labeled Button 1 and Button 2 in the diagram below.

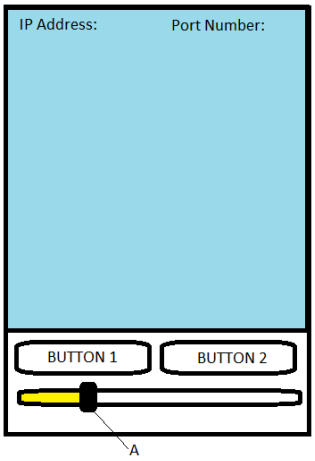
- **Search Button** – To use this, you simply tap the search button on the phone to activate the button. The search button will always be present when using the Android VRPN Controller as it is part of the hardware of the phone. (Not depicted in the diagram but is labeled as a magnifying glass on Droid phones)

- **Sliders** – To use the slider you will place and hold your finger on top of the small square labeled 'A' in the diagram below. You will then drag your finger left or right along the slider to move the box's position.

- **Tracking Box** – On the screen there may be a large blank area that doesn't contain any buttons or sliders. This is considered the tracking area and is used by simply placing your finger anywhere within the box, holding it down on the screen, and dragging it in different directions. This may seem as though nothing is happening from the phone's perspective but it will be changing values in the virtual world. The tracking box is shown in blue in the diagram below.

- **Accelerometer** in the phone – This feature is not displayed on the phone but instead is built into the phone. To activate this, simply lift, lower, or move the phone side to side. The accelerometer inside the phone will pick up these movements and transfer the data to the virtual world.

- **IP Address and Port Number** – The user will not need to worry about these labels and numbers. These are displayed for and used by the Administrator creating the Virtual World.



DEVELOPER MANUAL

Creating Controller Applications Using Our Compiled Library

A primary goal of our implementation was to minimize the work involved in creating custom VRPN Controller apps for the Android. Developers should be able to use the `JniBuilder` and `JniLayer` classes with their own applications in order to very easily send data from their app to a VRPN client app.

A possible issue is that of connectivity. By default, Android holds onto ports for two minutes after the application releases them. This can be a problem if the connection is interrupted. This issue should be resolved by code within VRPN but could potentially be a problem.

Working With the Java Classes

The `JniBuilder` class is used to tailor the `JniLayer` to your application's needs. Most data can be sent through VRPN in the form of a Button or an Analog. The application has one button server, which can be used with any number of buttons. There can be multiple analog servers, each with a user-specified number of channels. For example, multi-touch data can be attached to its own two-channel analog server. After using `JniBuilder` to specify the buttons and analogs to be used by the application, the `JniLayer` is produced using the `toJni()` method.

The `JniLayer` is responsible for communication with the compiled VRPN library. It has methods for updating the values for the buttons and analogs which were specified before its creation. The `mainloop()` method needs to be called at least once every

three seconds in order to prevent the client from complaining. Any time that the app is backgrounded, closed, etc, the `stop()` method should be called on the `JniLayer`. Similarly, the `start()` method should be called when the app comes to the foreground. We placed these calls in the `onPause()` and `onResume()` methods, respectively.

In addition to these two primary classes, we have implemented a few listener classes. These are not critical and can be omitted in custom apps.

Compiling VRPN on Android to Extend Functionality

You will need:

- Red Hat Cygwin Shell (and its associated programs): http://www.cygwin.com/
- Eclipse (We used Helios): http://www.eclipse.org/downloads/
- Android SDK: http://developer.android.com/sdk/index.html
- Android NDK - The standard version does NOT have support for STL, which is necessary in order to compile VRPN. Fortunately, support has been added by Dmitry Moskalchuk, who has a version of the NDK available here: http://www.crystax.net/android/ndk.php. We used r4.
- Java JDK (Obtained with Eclipse)
- Telnet – used for port forwarding to test network apps on the Android Emulator. It is disabled by default in Windows 7 but can be enabled in Control Panel > Programs and Features > Turn Windows Features On or Off.

Setup

Some changes need to be made to the CrystaX makefiles in order to compile VRPN. In the CrystaX folder, under `build/toolchains/arm-eabi-4.4.0`, open the `setup.mk` file and change the following lines:

```
TARGET_CC       := $(TOOLCHAIN_PREFIX)gcc
TARGET_CFLAGS   := $(TARGET_CFLAGS.common) $(TARGET_ARCH_CFLAGS)
```

to:

```
TARGET_CC       := $(TOOLCHAIN_PREFIX)g++
TARGET_CFLAGS   := $(TARGET_CFLAGS.common) $(TARGET_ARCH_CFLAGS) -D__ANDROID__
```

The first line forces interpretation of the *.C files as C++ source. The second adds a compiler flag which is used by `#ifdefs` throughout the VRPN source.

Project Setup

The Android NDK compiler looks for sources in a directory named `jni`. By default, it exports libraries to `libs/armeabi`. When an Android project requests that a library be loaded, the default directory is of the same name, under the Eclipse project directory. Therefore, a logical way to set up the project is to create a project in Eclipse and add a `jni` folder to contain the C++ sources.

Steps to Compile Native Code for Android

- Create a Java class containing native method declarations. These declarations include the native keyword. The class should also contain a request to load the exported library. The `JniLayer` class can be used as an example.
- It is helpful to have the directories commonly used throughout this process saved as environment variables. I will refer to the following:
    - `$PROJECT` – The base directory for the Eclipse project
    - `$JAVA` – The Java SDK binary directory (location of javah)
    - `$CRYSTAX` – Base directory of the CrystaX version of the Android NDK (location of ndk-`build`)
- Open Cygwin and navigate to the project directory. Use javah to create a C header from the java class, using the fully-qualified class name:

```
cd $PROJECT
$JAVA/javah –o jni_layer.h jni.JniLayer
```

  This will create a `jni_layer.h` file with method headers for the functions declared in the Java class. If no output directory is supplied, `javah` will output to the bin directory. The header file needs to end up in the `jni` directory.

- Implement the methods declared in the new header file.
- Create a makefile in the `jni` directory. This is what ours looked like:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := vrpn

LOCAL_SRC_FILES := \
       vrpn_Connection.c \
       vrpn_BaseClass.c \
       vrpn_Button.c \
       vrpn_Event.c \
```

```
        vrpn_Shared.c \
        vrpn_Analog.c \
        vrpn_Tracker.c \
        vrpn_Analog_Output.c \
        vrpn_Imager.c \
        vrpn_FileConnection.c \
        vrpn_Serial.c \
        vrpn_Android.c \
        jni_layer.cpp \

include $(BUILD_SHARED_LIBRARY)
```

- In Cygwin, navigate to the project directory and run the ndk-build command:

```
cd $PROJECT
$CRYSTAX/ndk-build
```

- If the compile is successful, a library should be exported to the `$PROJECT/libs` directory. The library should be ready to use.

Redirecting `stdout` to the Android Log

It is extremely helpful to see debug messages from the C++ side in the Android Log. It is possible to redirect `stdout` to the Android Log by creating (or editing if one exists already) a `local.prop` file on the emulator. This can be done by creating a text file containing the following line:

```
log.redirect-stdio=true
```

Save it as `local.prop` and move it to the data folder shown in the File Explorer in Eclipse while in DDMS Perspective.

Testing with Telnet

In order to connect to the emulator from a test program, it is necessary to do some port forwarding. We use telnet. With the emulator running:

- Open a command prompt
- Type:

```
telnet localhost 5554
```

- The Android console will open. Set up a redirect using:

```
redir add (tcp|udp):xxxx:yyyy
```

where (tcp | udp) is the protocol to use (we currently only support TCP), xxxx is the host port, and yyyy is the port on the Android. For example, we used:

```
redir add tcp:5000:3883
```

Notes

- Eclipse is slow to recognize that a library has changed. It is often necessary to clean the project and restart the emulator. It is helpful to get into the habit of closing the emulator, cleaning the project and re-running whenever the library is replaced.

- Although the current implementation only utilizes JNI calls from Java to C++, it is certainly possible to do the reverse. This would allow message-passing to the Android and could be used for a number of expansions on this basic server.

- Android uses the Arm architecture, which may or may not use swapped byte order and word order, depending on the device hardware. On our test devices, the byte order was swapped but the word order was not. If data seems to mysteriously change or disappear, the first place to look is in `vrpn_Shared.C`, in the function `htond()`. Note that when compiling Android libraries, both `__arm__` and `__ANDROID__` will be defined. As it stands, the word order swapping function is ignored if `__ANDROID__` is defined (line 196). On other devices, it is possible that the word order will need to be swapped, and that `&& !defined(__ANDROID__)` needs to be removed. A better solution would be to automatically detect the byte and word order through the use of doubles with known bit patterns.