

# Bluetooth “Clicker”

---

Response Pads for large classes,  
using Bluetooth devices

**Dana Goyette**

**June 2010**

Senior Project  
Computer Engineering, B.S.

## Acknowledgements

I would like to thank Professor Oliver for being my senior project advisor / mentor, for giving me this interesting project, and for purchasing the BeagleBoard and the Bluetooth adapters I used for my project development. I would also like to thank my parents for helping me stay on top of my work during this project (as well as during all other quarters). For software tools, I would like to thank the DBus-C++ developers for the tool that makes DBus communications simple, and Leor Zolman for creating the sanity-saving STL filter for C++.

## Introduction

When teaching classes with many students, many teachers opt to use “Clicker” devices to electronically poll students for answers to quiz questions. Though these devices are useful, they are not cheap for students. My project aims to replicate the function of these “Clicker” devices using standard Bluetooth devices many students already own. The overall goal of this project is to create a framework for two-way communications between a base station and multiple remote devices, presenting a simple console interface for applications.

## Background

When teaching classes with large number of students, the last thing any teacher wants is to have to grade quiz papers – so instead, teachers opt to use “clickers” (response pads) to poll students for answers. For the teachers, the clickers are a real win – collecting quiz answers electronically means they save a lot of time, and they only have to buy the receiving “base station” once.

For students, however, the clickers are inconvenient. The Clicker devices cost around 20 to 30 dollars, and then some brands demand additional money for activation – and then different classes may require the purchase of different brands of Clickers. Finally, once the course ends, the Clicker device is useless.

This project aims to replace the proprietary response pad (Clicker) devices with standard Bluetooth devices. Nowadays, most students have Bluetooth-enabled phones, so why not use those to get people's responses? This will be free for those students that already have smartphones.

The existing Clicker devices exist primarily for one-way communications: the teacher displays questions on the projector, and students answer the questions with their clickers. This project aims to improve upon that, by allowing the base station to broadcast messages to all remote devices. This would enable the cellphones to display the questions, and perhaps even run text-to-speech on them.

## Requirements

- The project should use Bluetooth for communications. The use of existing Bluetooth devices will allow students to not spend any money on new hardware.
  - The design initially called for using keyboard (HID) devices; after such devices presented problems, the project changed to using RFComm sockets.
- The base station running the software must be able to communicate with a large number of devices simultaneously.
  - This project aims to replace the Clickers, which are often used in classes of over 100 students.
  - Bluetooth claims a maximum of 7 slave devices active at once, but does not clearly define the length of a time slice.
  - For very large number of devices, multiple Bluetooth adapters may be required.
- The base station should automatically pair with and connect to devices. This will prevent the software from cluttering the teacher's host computer with a large list of Bluetooth devices.
- The project should establish two-way communications between the teacher and the students. Specifically, the base station should broadcast the teacher's messages to all remote (student) devices, and should return to the teacher the responses sent by each student.

## Tools

### Hardware

When deciding what hardware to run my project on, I aimed for several overall goals: the device should be inexpensive, low-power, small, and standardized enough to run Linux on without having to spend a long time figuring out how to bring up the board. I considered using Agilent's ARM9 products, but even after the 25% discount John Oliver receives, the base price of at least \$500 put Agilent's boards out of the desired price range. After some research, I came up with the following table of candidates:

Brand	BeagleBoard	Gumstix	PlugComputer	Complab	NorhTec
Board	BeagleBoard	Overo Air	SheevaPlug	Fit-PC2	Microclient 2
Minimum Price	\$150	\$150	\$100	\$200	\$190
Processor Type	TI OMAP3	TI OMAP3	Marvell Sheeva	Intel Atom	SiS550
Instruction Set	ARMv7	ARMv7	ARMv5	x86	x86
CPU Frequency	720 MHz	600 MHz	1.2 GHz	1.6 GHz	200 MHz
Dimensions	79 x 76 mm	58 x 17 mm	110 x 70 mm	120 x 116 mm	115 x 115 mm
RAM	256 MB	256 MB	512 MB	1024 MB	128 MB
Onboard Flash	256 MB	256 MB	512 MB	4096 MB	Unspecified
Memory Card	SD	Micro-SD	SD	SD	CompactFlash
USB Host	1	**	1	4	2
Serial	Header	**	USB-Serial	Dongle	2
Network	USB	BT and Wifi	Gigabit	Gigabit	100 Megabit
Video	HDMI, S-Video	**	No	DVI	VGA
Audio	In/Out	**	No	In/Out	In/Out
Power Usage	2 W	2 W	2.3 – 7 W	6-8 W	4 W
Other		** adds \$50-\$80	3-6 weeks to ship		

In the end, I settled on the BeagleBoard – it had all the features I needed, and nothing unnecessary, for the most reasonable price and availability. The best part about using this board is that Ubuntu 10.04 (“Lucid Lynx”) officially supports it. The nearest competitor, Marvell, lacks the ARMv7 support required by Ubuntu 10.04.

Bringing up the board was simple: I just downloaded and extracted a root file system image and a kernel to an SD card, inserted the card, and plugged in the board. From there on, I was able to treat it just like any other Linux system – no need to deal with a special embedded OS.

For Bluetooth, I initially looked into various special-purpose adapters, such as SDIO (SD Card form factor) Bluetooth cards, but nearly all these cards would have required using the BeagleBoard's expansion header, were not available in Class 1, or were outlandishly expensive (at least \$50). Since the BeagleBoard has USB host capabilities directly, I settled on inexpensive Class 1 USB Bluetooth adapters.

### Software

When I started on this project, I knew I would need to talk to the Linux Bluetooth stack (BlueZ) over DBus, but it took me a while to figure out what tools and language to use for this. DBus is an inter-process communications standard that allows applications to communicate, even from users to software running as root. Applications using DBus can make and answer calls to various interfaces, using object paths and interface names. One of the developers of the Linux network-client framework, NetworkManager, gives a good analogy for D-Bus, using a fast-food restaurant as an example (Williams, 2010).

I initially attempted to use the DBus Glib bindings (in C) to communicate with Bluez, but after a few days of wrestling with poor documentation – for example, documentation stated that some methods returned an “object”, without explaining what data type, exactly, this “object” is in a non-Object-Oriented language – I decided that the object-oriented nature of DBus demands a truly object-oriented language.

After considering whether I’d need to learn Python (the language many DBus examples are written in), I found the DBus-C++ bindings. This tool made it simple to interact with Bluez’s objects over DBus, by abstracting them as local object. For example, the Bluez daemon presents DBus objects for all Bluetooth adapters and devices.

One last tool I have found essential for coding in C++ is Leor Zolman’s `STLfilt` – “An STL Error Message Decryptor for C++” (Zolman, 2008). This tool takes the cryptic, overly-verbose textual spew g++ gives, and filters it down to something sane. For example, it can trim “no match for function()” errors from multiple lines of candidate-functions spew, down to just one line of what function was not found.

## Design and Testing

### Using DBus-C++

My first task in coding my project was to figure out how to speak to the Bluetooth daemon over D-Bus, with the goal of getting the Adapter to start scanning for devices and telling me when it found them. Online documentation for the DBus-C++ libraries set of libraries is sparse; my main source for sample code was the source of the `libdbus-c++` package (“`apt-get source libdbus-c++`”).

It took me a while to figure out how to use the DBus-C++ bindings. The DBus-C++ bindings, essentially, create local objects to represent objects you want to interact with over DBus. Creating these objects requires the following process (using an Adapter object as an example):

1. Introspect (inspect) the object you are interested in. This generates an XML description of all interfaces and methods on the object.
  - a. Tool used: `dbusxx-introspect`
  - b. Input: `Path /org/bluez/<pid>/hci0`
  - c. Output: `hci0.xml`(I then trimmed `hci0.xml` to create an `adapter.xml` file with only the Adapter interface.)
2. Automatically generate an “InterfaceProxy” from the XML file:
  - a. Tool Used: `dbusxx-xml2cpp`
  - b. Input: `adapter.xml`
  - c. Output: `Adapter_proxy` class (an InterfaceProxy for `org.bluez.Adapter`)
3. Manually create a class to handle the object. The intersection of the interface name and the object path specifies what events your code receives:
  - a. InterfaceProxy: `Adapter_proxy` (created above)
  - b. ObjectProxy: `Path /org/bluez/<pid>/hci0`
  - c. Created class: `Adapter`

This last class, the `ObjectProxy`, defines nearly all of my code.

## Finding Devices

After telling the Adapter to start scanning for devices, I had to figure out what to do with each device it found. I decided to have each Adapter pass its devices on to the Manager, so the Manager could load-balance devices using multiple adapters. Assuming a limit of 7 active devices at once, the Manager class watches until a device has been seen more than once, and then pairs it with the adapter that has the lowest sane signal strength, to avoid using up all the strong connections. However, my tests in the Capstone lab later showed this load-balancing to be unnecessary, at least for the number of devices on hand.

## Pairing Devices with Adapters

The act of pairing with a device required me to implement an Agent object to give a passcode. The Agent class is an “ObjectAdaptor,” which is the converse of the “ObjectProxy” classes. A **Proxy** lets your program act upon other DBus members; an **Adaptor** exists for *other* DBus members to act upon *your* objects. To simplify pairing, I made my Agent always return an integer of 0 and a string of “000”.

In implementing the Agent, I also discovered an oddity in the BlueZ DBus interface: the “RequestConfirmation” method is defined as a *void*, yet expects an Agent to *return* an error if the user denies authentication – and the DBus-C++ wrappers don’t handle this odd case. Thankfully, my software never denies.

When pairing, I initially ran into a deadlock situation: the act of pairing is like knocking on a door, and then having somebody pop out from a window, somewhere else, to ask for the passcode. When my program remained standing at the door – that is, by making a blocking call to the CreatePairedDevice method – it never received the passcode request. Once I made the CreatePairedDevice call non-blocking, my program successfully returned the passcode and paired. The DBus-C++ library defaults to blocking calls; making non-blocking calls required defining my own methods.

## Connecting to Paired Devices

### *Initial Design: HID devices*

Once I had paired with a device, it came time to connect to services on the devices. In my Adapter object, I created a thread that looped through devices, periodically connecting and disconnecting each device as a way to deal with the limit of 7 devices active at once. After implementing this, I soon discovered that my initial target, a HID (Human Interface Device) keypad, behaved badly – it refused to reply to any requests to connect it (“Host is down”) unless you put it in Discovery mode, even after you had paired with it. In addition, a successful connection to a HID device did not report where (in /dev/input/) the device had appeared – so I would need a separate tool to discover this.

### *Revised Design: RFCOMM sockets*

Faced with an uncooperative keypad device and the need for another new tool, I decided to switch to RFCOMM sockets. When I initially just told BlueZ to connect to a Serial Port, it created a “/dev/RFCOMM $N$ ” file and told me this path; however, this call tended to be slow, sometimes taking upwards of 5 seconds. I eventually figured out that I can just directly open an RFCOMM socket – this is the function of my Socket-Handler class. Unfortunately, the Boost Socket libraries seem to handle only TCP, UDP, or Unix Domain sockets, and C++ gives no obvious way to bind a bare file-descriptor or FILE\* to C++ streams; thus, I had to implement socket I/O in C.

## Finding RFCComm Channel

When connecting to an RFCComm socket, an initiating device needs to have the channel or port number that the RFCComm service is on; unfortunately, the Dbus interface for doing so is prone to hangs, and returns a cryptic XML service-record format (please see Appendix A: XML Service Record Format). Using this XML would have required choosing and learning to use an XML parser. Rather than wrestling with XML, I initially decided to use an inelegant method to find the channel number: I used `popen` to capture the output of the system utility “`sdptool`”, and looked for the string indicating the channel number. Unfortunately, this method tended to fail with “Too Many Links”. I later implemented a far more elegant method, based on sample code that uses Service Discovery Protocol directly (“`slzckboy`”, 2007).

## Testing Multiple Devices

Once I had my program reading from RFCComm sockets (with an RFCComm server app on the other end), I had to test how well that would work with multiple devices active at once. The Bluetooth standard advertises that only seven devices can be active at one time, but did not plainly specify how small the slices of time were. I went to the Capstone lab to test with the Bluetooth adapters Professor Oliver purchased for me, one adapter per computer. I was glad to discover that the Bluez daemon automatically handled at least 10 devices (HID and Serial) at once. This meant that I would not have to deal with the time-multiplexing of connections myself.

## RFComm Communications

Once I had verified that multiple devices worked automatically, I dropped my “disconnect” code from my Adapter class, and went on to refine the socket communications. The functions of the base station and the remote devices are simple, though the *remote* devices are the ones that `listen()` for incoming connections.

- Remote Device:
  - Listens for incoming connections
  - Sends standard input to the base station
  - Prints messages from base station to standard output
- Base Station
  - Connects to Remote Devices
  - Reads from each Remote Device
    - Prints MAC address and message to standard output:  
[00:02:72:a9:2b:03] «Hello!»
    - Echoes message back to sender
  - Reads from standard input and broadcasts to all Remote Devices

During my testing, even with just two devices, the RFCComm sockets tended to lose data. I initially concluded that RFCComm was unreliable, and documented interference issues as one area needing more work.

Later in the quarter, in my Networks class (CPE464), I encountered similar losses with TCP sockets: my program was losing all data following a newline character. As it turns out, my program was using `fgets()`, which breaks in exactly this manner when used on a non-seekable file-descriptor such as a socket. During Finals week, I made the RFCComm sockets perfectly reliable by implementing my own `GetLine` function, using character-by-character buffered reads with `fgetc()`.

## Other issues

When debugging the DBus-C++ objects, I very frequently needed to print `DBus::Variant` objects. A `Variant` may contain any of many various data types (such as strings, or signed and unsigned integers of several lengths), or even an array of other `Variant`s. When I tried to directly print a `Variant`, the compiler complained about ambiguous functions; to fix this, I had to make my own function that accepts a `Variant`, and uses the DBus type signature to print the correct type. For example, if `var.signature() == 's'`, then it is a string, so I call `get_string()` on the “reader” iterator for that variable. Likewise, setting values requires using a “writer” iterator.

Throughout my use of the `ObjectProxy` classes, I kept running into some odd hang on some method calls – `strace` would show my process making a request and getting a full response, but then my app would hang on some `futex` call (short for “fast userspace mutex”). My only workaround was to avoid making blocking calls on functions that tended to hang. For example, instead of having the `Adapter` call `CreateDevice` and the `Device` object call `GetProperties`, I fed the properties seen at `DeviceFound` time into the `Device` class’s constructor, and made the call to `CreateDevice` non-blocking. Over the summer, I plan file a bug report about this hang.

While working out the automatic pairing, I ran into some odd behavior, both with Cambridge Silicon and Broadcom Bluetooth adapters: under heavy new-connections load, the Bluetooth adapter would momentarily unplug itself from the USB bus – and when it returned, BlueZ would report an error (“Could not find matching adapter”) and refuse to use it. Additionally, my program hits an assertion failure when an adapter is removed.

## Design Summary

My software can be summarized by describing the main classes involved.

- **Adapter:**
  - Discovers devices, creates `Device` objects, and tells `Manager` what it’s found.
  - Iterates through list of devices, checks for Serial Port, and tells `Manager` to connect.
- **Manager:**
  - Creates `Adapter` objects, and pairs devices with `Adapters`.
  - Creates `Socket-Handler`, and tells it when to connect.
- **Agent:**
  - Replies to passcode requests (integer 0 or string “0000”); tells `Manager` when it’s done
- **Device:**
  - Discovers and keeps track of all services a device supports.
  - Creates `SerialDevice` object to keep track of `RFComm` channel.
- **Socket-Handler:**
  - Maintains list of active sockets.
  - Reads from and writes to sockets, as described in `RFComm Communications`, above.

## Conclusion, and Future Work

Since this project has reached the stated goal of creating two-way communications over Bluetooth, we can now create future projects based upon this foundation. Such projects may be either enhancements to the existing code, or implementation of new applications on top of this code.

- One issue: teachers don't want students to have internet access to look up answers online.
  - One solution: make the response-pad application block out internet access during quizzes.
- Some students still have simpler Bluetooth phones; this will need separate solutions:
  - Implement a responder in languages such as Verizon's "BREW".
  - Integrate text-message (SMS) receiving into the program.
  - Allow the use of Bluetooth keyboards (HID devices).
  - Create simple Bluetooth-Serial keypads.
- Test the project with a much larger number of devices
  - My tests were all with fewer than 20 devices; large classes often have over 100 students.
- Refinements to this project's source
  - Track down and help fix the Dbus-C++ hangs, and then switch calls back to blocking.
  - Decode the XML service record Bluez returns over Dbus, rather than using SDP directly.
  - Fix the assertion failure that occurs when a Bluetooth adapter disappears.

This project gave me an opportunity to learn how to use Dbus, as well as learning how to use sockets. I also learned how the Bluetooth standard works, and how devices pair. I also gained useful experience with C++ and the Standard Template Libraries, and learned how to handle locks when multiple threads need to share data.

## Works Cited

- "slzckboy". (2007, March 23). *searching for bluetooth services using bluez sdp*. Retrieved June 4, 2010, from LinuxQuestions.org: <http://www.linuxquestions.org/questions/programming-9/searching-for-bluetooth-services-using-bluez-sdp-539873>
- Maemo Community. (n.d.). *Maemo 5 API Documentation: Bluez*. Retrieved June 10, 2010, from Maemo.org: [http://maemo.org/api\\_refs/5.0/5.0-final/bluez/](http://maemo.org/api_refs/5.0/5.0-final/bluez/)
- Volz, A. (2010, May 8). *DBus-C++*. Retrieved June 10, 2010, from Sourceforge.net: <https://sourceforge.net/projects/dbus-cplusplus/>
- Williams, D. (2010, May 7). *Eat Burgers on the Short Bus*. Retrieved June 10, 2010, from <http://blogs.gnome.org/dcbw/2010/05/07/eat-burgers-on-the-short-bus>
- Winterbach, W. (2007, December 19). *Compiling a C++ DBus program*. Retrieved October 2009, from Apertium wiki: [http://wiki.apertium.org/wiki/Compiling\\_a\\_C%2B%2B\\_D-Bus\\_program](http://wiki.apertium.org/wiki/Compiling_a_C%2B%2B_D-Bus_program)
- Zolman, L. (2008, January 28). *STL Error Decryptor for C++*. Retrieved April 14, 2010, from BD Software: <http://www.bdsoft.com/tools/stlfilt.html>

## Appendix A: XML Service Record Format

The desired value, the RFComm Channel, is 0x10.

```
<uint8 value="0x10" />
```