BoardKey

A keyboard simulator for macOS

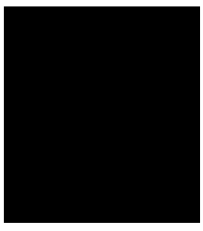
University of California, Los Angeles
CS/EE M117 Spring 2018

Professor Dzhanidze

June 7th, 2018

GitHub Repo
https://github.com/ckwojai/BoardKey

Andrew Gao
Bryson Li
Eric Oh
Joel Mashita
Wilson Chang



Abstract

This project explores a wireless technology, namely Bluetooth, which serves as the main data transmission method over small distances for modern consumer devices. The goal was to create an iOS mobile application that allows the iPhone to be used as a wireless Bluetooth keyboard for macOS. Two Apple frameworks, Core Bluetooth and Core Graphics, were utilized to establish a connection between the iPhone and Macbook and simulate tap events from one platform as hardware keystrokes in the other. The results show that the process of creating such an application divulges many overlooked layers of abstraction beneath these technologies that offer far more functionality than advertised.

1. Motivation

It is eight in the evening. You and your family sit down on the couch to watch your TV that is connected to your laptop as an the main input. You use your wireless keyboard to type into the search bar, but suddenly, the battery dies. You then hunt for batteries only to find that there are none. Perhaps you could use your external keyboard but your laptop USB ports do not support the peripheral device without the proper dongle.

"There has to be another way," you say to yourself. "What if I could use my smartphone as a wireless Bluetooth keyboard?" You now can if you are on board with using BoardKey - a keyboard simulator app for macOS systems with Bluetooth technology. With BoardKey, a possibility is made a reality!

2. Functionality

Coded with the newly popular and robust language Swift, BoardKey uses a very simple model-view-controller framework with two views, one for the main Bluetooth pairing process and another for the interface of the keyboard which will accept the input from the user.

In order for the application to function properly, the user must turn on Bluetooth and open the desktop software. Once paired, the phone will act as a Bluetooth keyboard. The exact Bluetooth and Apple technologies will be covered in depth in subsequent sections.

3. Drawing Board

As with any design process, the functionality of the application was compartmentalized to separate backend front-end and back-end functionality. To complete our task, these were some individual goals that served as a guideline for dividing work among the group members as well as milestones to track progress.

1. Create virtual keyboard UI for smartphone application.
2. Simulate key press events from the virtual keyboard UI from the smartphone.
3. Establish Bluetooth connection method between the smartphone and the laptop.
4. Accept the keyboard characters from the smartphone and register them to the laptop.

4. Key Technologies Utilized

The main IDE used for development was XCode 9.2 on macOS High Sierra 10.13.4. The primary language used for development was Swift 4 in order to build for iOS 11.3.

In the Appendix, we delve into the exhaustive details regarding the classes and methods of the Core Bluetooth and Core

Graphics frameworks that are utilized within the implementation.

An option is at one's disposal to skip the Appendix entirely and refer back to it when necessary; however, the code review will be greatly facilitated and the implementation, elucidated should one choose not to do so. In order to maintain fluidity when reading, Appendix sections A1 and A2 are recommended readings after sections 4a and 4b respectively.

4a. Wireless Technology: Core Bluetooth

The Core Bluetooth is an Apple framework and library that allows communication between iOS and Mac apps using Bluetooth. The connection is between two identities with a relationship that is displayed in Figure 1.

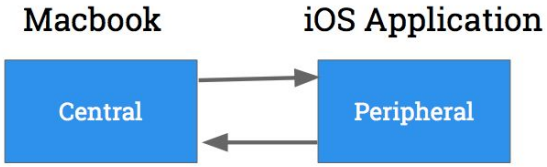


Figure 1: Central and Peripheral

The Peripheral acts likes a server, constantly broadcasting the information it can provide in the form of advertising packets which are small bundles of data containing information and data about the peripheral device regarding the name and primary functionality.

On the other hand, the Central acts like a client as it scans for Peripherals, establishes the connection with the desired peripheral and listens to data from it.

4b. Core Graphics

Core Graphics is an Apple framework, based on Quartz drawing engine which provides 2D rendering. This framework is used for handling drawings, image creation and editing, and image display. In the macOS, the framework provides services which allow for working with display hardware, handling user input events, and managing the windowing system.

5. Implementation

The implementation of BoardKey is of two parts: one for the Peripheral, with the code located in the IOSPeripheral directory, and another for the Central, with code located in the MacCentral directory. Both implementations can be found in the GitHub repository and their overviews explaining each class and method are covered in detail in the Appendix. Appendix sections A3, A4, and A5 are recommended readings right after this section.

5a. Connection Establishment

For the connection implementation, the main framework used was the Core Bluetooth framework. For the Peripheral, we initialize the CBPeripheralManager along with the CBPeripheral. CBServices are initialized for the peripheral and a CBAdvertisement instance is created and advertised to possible Centrals.

On the Central side, we perform a similar procedure by initializing a CBCentralManager and call its scanForPeripherals method. This will allow for the Central to scan for and connect to our desired Peripheral with an identifier which we have hardcoded for the sake of simplicity and efficiency. After connecting to the Peripheral, the CBCentralManager will listen for additional advertising packets which they will parse when received. The Central manager will then search for their services and subscribe to them via setNotifyValue method to their CBCharacteristics.

5b. Keystroke Input Processing

For accepting events of user keyboard inputs, the main frameworks utilized were the Core Graphics framework as well as the UIKit.

On the Peripheral side, a user interface replicating the QWERTY layout was designed with each keyboard button of class UIButton. The key when pressed updates the keyboardState value to the text value of the tapped button via the peripheral manager's updateValue method. This method will send the updated characteristic value, the new key input, to the subscribed Centrals via a notification.

The central manager detects when this new characteristic value has been received, extracts the keyboard input value from the characteristic, and calls the executeKeypress method with the output of the str2code function for the input value as the argument.

This completes the process of taking the input from the Peripheral, our smartphone, and perform the corresponding hardware event on the Central, the macOS. When the peripheral is stopped, the peripheral manager and its services are deleted.

6. Discussion

Setting up the Bluetooth connection between the phone and the computer was simple, but it had to be hardcoded. Thus, we were unable to use different computers or phones to use the app with our current implementation. Using a searching method would allow us to accomplish this.

Simulating keystrokes through code on the computer was also straightforward once we found the key codes of each character. However, we were unable to find out how to use the Shift+Key functionality to modify inputs, for example, from lowercase to uppercase. We discovered that there are separate key codes for uppercase letters that could be used as a substitute for this feature. Our solution was to rather use a toggle system for the implementation.

We were able to convert the iPhone inputs on the app to keystrokes on the computer. With our implementation, the keystroke event does not occur until the virtual key is both pressed and released. This means that holding down a key does not generate a stream of keystrokes on the computer. This could be implemented by having a time buffer after a keydown event that, if exceeded, would start sending a stream of keystroke events on the computer.

7. Possible Improvements

For additional features, there are many that must be added for BoardKey to comprehensively fulfill a user's needs and be a viable product. The necessities of a consumer in the twenty-first century are in constant change as the methods of interaction with technology are continually revolutionized and modernized to complement our surroundings.

For the privileged, the surrounding may be a smart home setup. You adjust the WiFi lights to a reasonable brightness, turn off the smart fan to limit background noise, and proceed sit down on your couch to watch your smart TV, which is connected to a desktop rig as the main input source. To control this systematic architecture of computers, disguised as smart devices individually administering our daily needs, a gadget of some sort is undoubtedly needed. A smartphone could be used as a wireless Bluetooth keyboard and trackpad or a voice controller, newly popular on the current market, are also an alternative option. Indeed, the controllers of the previous technological generation have not been and will not be rendered obsolete. That is why the first feature to be added to BoardKey

alongside keyboard capabilities would be a touchpad option.

Replicating similar functionalities to those of the Apple trackpad, the touchpad would offer customizable gestures registering as events capable of desktop environment control for multitasking and increasing productivity.

A touch controller satisfies a special niche as it offers a degree of control that a keyboard or a conversational system may not due to its otherwise inherently unique-to-platform rendering of touch sensitivity. For the artistic endeavors of artists and creative minds, a touchpad is not only a necessary canvas but also a quintessential vehicle through which they manifest, with subtlety of touch, their imaginative spirit.

On the other side of the globe, in perhaps in a third world country, there is a society that has not yet experienced the technological revolution or used most glorious modern invention - the Internet. The best tool is the tool that you have with you and to them, the smartphone could be the first screen, the first computer, the first tool that comes into their possession. In order to empower those users to harness the full potential of a smartphone, it must emulate many key apparatus but more importantly be capable of enhancing and extending functionality to accommodate their otherwise contrasting needs.

For the non-privileged, the living room consists of a television that is considered vintage by first world. The elderly mother walks up to the antique in order to change the channel as the remote has broken years ago. Here, a smartphone with the proper universal remote UI could be used as a controller for the television after creating a signature IR signal to match the television. And thus additional features that could thus be added to BoardKey would be IR signal broadcasting and matching as well as various UIs, some customizable, for appliances and devices of previous generations.

8. Conclusion

The journey of creating this mobile application came with various challenges and learning experiences. As the application needed to communicate wirelessly with a computer, the choice for the main wireless technology to utilize for low range and low energy consumption was an obvious one: Bluetooth.

The Core Bluetooth framework was a joy to use and research as the procedural architecture uncannily exemplified a microcosm of the current world that stands devoid of information privacy. I did not learn the functionalities, but rather, the framework taught me. Thus, the methods and attributes of instances were easy to use from the get go.

The main obstacle we faced was implementing multi-key inputs. The workaround solution was to use a toggle system for upper-case letters and symbols. The feature of repeated keystroke inputs from holding the button was not included in this build. The challenge brought great insight on how user input on a touch screen is registered as events and processed in the device as well as the complications in implementation a simple change in the tap length brings.

For additional features, there are many that must be added for BoardKey to be a product that a consumer would want to use. Currently, it only fills a specific niche; however, there is no doubt that the smartphone will remain as the ubiquitous go-to controller for the masses and its future uses will change according to shift in necessity of the masses.

References

1. "Core Bluetooth | Apple Developer Documentation." https://developer.apple.com/documentation/corebluetooth. Accessed 29 May. 2018.
2. "Core Graphics | Apple Developer Documentation." https://developer.apple.com/documentation/coregraphics. Accessed 29 May. 2018.
3. "Bluetooth LE IOS Swift" https://github.com/ormaa/Bluetooth-LE-IOS-Swift. Accessed 29 May. 2018.
4. "CoreBluetooth Peripheral Example" https://github.com/liquidx/CoreBluetoothPeripheral. Accessed 29 May. 2018.
5. "Core Bluetooth Tutorial for iOS: Heart RateMonitor." https://www.raywenderlich.com/177848/core-bluetooth-tutorial-for-ios-heart-rate-monitor. Accessed 29 May. 2018.
6. "Mac-bt-headset-fix." https://github.com/jguice/mac-bt-headset-fix/blob/master/README.md. Accessed 29 May. 2018.
7. "MacOSX-SDKs." https://github.com/phracker/MacOSX-SDKs. Accessed 29 May. 2018.

Appendix

A1. Core Bluetooth Framework

The Peripheral is managed by, you guessed it, a peripheral manager. It is of the class CBPeripheralManager, a delegate class which adopts its methods and protocols from the CBPeripheralManagerDelegate protocol. The peripheral manager essentially manages local, and quite literal, peripheral devices, represented by object instances of class CBPeripheral, and their database of services.

Now what is a service? Well, it is somewhat similar to how it sounds, but more specifically, it is "a collection of data and associated behaviors for accomplishing a function or feature of a device" represented by the CBService class (Core Bluetooth Documentation). The peripheral manager initially broadcasts this service via the startAdvertising method to the Central and consequently every instance afterwards if the service has been updated to all Centrals in its subscription. This notification is sent as an article known as an advertisement.

Such procedure seems all too familiar and reminiscent of online advertisements that incorporate, dare I say it, *class-ifying* information that is unethically amassed by personal 'data broker' companies. Undeniably and inherently different in its regard for information, the peripheral manager's advertisement, an object instance of class CBAdvertisement, contains comprehensive information about the peripheral device's name, its primary functionality, and unique identifiers such as the UUID - all of which are public properties that are broadcast to all surrounding Centrals. All jokes aside, I hope that no one creates an application that advertises my online fingerprints.

And if this traversal down the class hierarchy wasn't already enough, these services have one or many characteristics. No, literally instances of characteristic objects of the class type

CBCharacteristic. “CBCharacteristic represent[s] further information about a remote peripheral’s service” and “contains a single value and any number of descriptors describing that value” (Core Bluetooth Documentation).

A2. Core Graphics Framework

The classes and methods from the Core Graphics framework utilized in our implementation are thankfully quite few. Low-level hardware events are represented by the class CGEvent, often caused by a source which is represented by an instance of class CGEventSource, which denotes the source type be it a keyboard key press or a mouse click. Lastly, the class CGKeyCode “represents the virtual key codes used in keyboard events” (Core Graphics Documentation).

A3. BLEPeripheralManager.swift

The Peripheral implementation located in the BLEPeripheralManager.swift file contains the main class of BLEPeripheralManager which extends the methods and protocols of CBPeripheralManagerDelegate and NSObject. The BLEPeripheralManager class contains one of each instance of CBPeripheralManager, CBPeripheral, CBService, and CBMutableCharacteristic. The self-defined local variable to accommodate our needs is keyboardState in order to keep track of the keyboard input received from the user. Other notable variables are global values that are unique identifiers of the peripheral such as peripheralName, CH_READ, and CH_WRITE.

Custom user defined functions in this class are startBLEPeripheral, stopBLEPeripheral, createServices, and getState; the functions initialize the peripheral manager, stop the peripheral manager and remove the services, initialize service instances when called in the constructor, and track the manager state, respectively.

A4. BLECentralManager.swift

The Central implementation located in the BLECentralManager.swift file contains the main class of BLECentralManager which extends protocols and methods of CBCentralManagerDelegate, CBPeripheralDelegate, and NSObject. The BLECentralManager class similarly contains one of each instance of CBCentralManager, CBCentral and CBPeripheralManager.

The self-defined functions in this class are startBLECentral, stopBLECentral, and getState; the functions initialize the central manager, stop the central manager and track the manager state, respectively.

A5. Tools.swift

The Central implementation directory also includes an additional library of self-defined functions in the Tools.swift file. The functions included in this file are keyboardKeyDown, executeKeypress, and str2code; the functions initialize the event and its event source and executes it, make a call to keyboardKeyDown with the converted CGKeyCode value from the string input, and convert a string input to its corresponding CGKeyCode value, respectively.