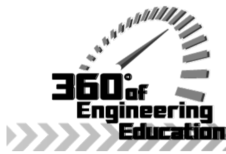

121st ASEE Annual Conference & Exposition
Indianapolis, IN
June 15-18, 2014

Paper ID #9485

Bluetooth Smart Phone App for Terrain Sensing Vehicle

Dr. Mohammad Rafiq Muqri, DeVry University, Pomona
Mr. Brian Joseph Lane

Brian Lane is a Project Engineer at inhouseIT, a managed service provider focusing on small to medium sized business. He received his A.S. in 2009 in Electronics and Computer Technology, continuing on to get his B.S. in Computer Engineering from DeVry University. His interests include programming applications for any Apple devices and learning new programming languages.

©American Society for Engineering Education, 2014

Bluetooth Smart interface for Terrain Sensing Vehicle

Abstract
How we communicate usually determines the success with which our message is received, understood, or even accepted. Many universities are faced with the challenges of attracting new high school graduates, engaging them, and enhancing their skill-set. Modern day students prefer multi-tasking, learn better in social contexts, and thrive on smart phone apps and cyberspace. These smart phone apps can be used as a medium to expose students to nontraditional topics not covered in conventional software courses.

As most of today's technology is heading toward wireless communication, new techniques must be developed to effectively design and create applications that are able to use this wireless technology. Students at our university have designed and tested a Terrain Sensing Vehicle. The purpose of this paper is to develop an iPhone application that can connect with a Terrain Sensing Vehicle via Bluetooth technology and control the vehicle from a remote location, allowing the user to change settings on the fly. With this smart phone application, it will be possible to control the vehicle from up to 30 feet away. It is expected that this application will help reduce the direct physical interaction required in traditional settings between the vehicle and a human. The vehicle will be able to be placed anywhere and will enable user to remotely control and change the vehicle operating modes.

The intent of this paper is to document our experience in designing and operating the student's capstone senior project contest. As such this paper concerns problem solved and lessons learned while conducting this design contest. The goal of this paper is to pass on information useful to anyone contemplating related work, where similar occurrences are likely. This paper will attempt to demonstrate that building smart phone applications is not just limited to games, but students can also use build applications that inform and educate.

Introduction

The objective of this paper is to document our experience in designing and developing the smart Bluetooth interface for an Autonomous Terrain Sensing vehicle (ATSV) which was designed for the computer engineering capstone senior project. As such this paper concerns problem solved and lessons learned while conducting the design and testing; the paper also describes the first hand experiences of the student who integrated and developed the Mac Bluetooth interface. One of the downsides of typical mobile robots is that they can't travel safely over rough unknown terrain. The key feature of this ATSV is that it can travel over uneven and vegetated ground while remotely-controlled.

Autonomous Terrain Sensing Vehicle has three different settings that a user can select physically from the keypad on the HC12 board or from a GUI that has been designed to communicate via Bluetooth. The vehicle has sonar that detects if there is an object ten inches from the front of the vehicle; when the sonar detects an object it stops, reverses, rotates to the left, and continues at half speed for one second and then returns to full speed again. The vehicle has a track configuration for many types of terrain. Setting 1 starts with the tracks in the lowered position for smooth terrain while input is taken from the sonar. Setting 2 has the

tracks in a raised position for rough terrain with input also taken from the sonar. Setting 3 starts out with the tracks in a raised position and takes input from the accelerometer mounted to the vehicle. The vehicle then adjusts the tracks left, right, up, or down depending on the x and y axis thresholds. Like the other settings, input is taken from the sonar to detect object.

This experiment was done to integrate Bluetooth technology into the Terrain Sensing Vehicle. The work on this integration started with the development of an iPhone application that had all the buttons and functions working correctly with the exception of the Bluetooth. Due to the proprietary nature of iphone development and the availability of Foundation Framework for the Mac OS X, it was decided to make use of the internal Bluetooth chip from the MacBook Pro. As a result the Mac OS Bluetooth interface application was being designed and tested right from the Mac Pro Book without having to pay for Apple's iphone App development iOS Simulator. Appendix A contains the Bluetooth Project Class Definitions and Pairing Methods.

The Cocoa Environment
Cocoa is a set of object-oriented frameworks that provides a runtime environment for applications running in Mac OS X and iOS. Cocoa is the preeminent application environment for Mac OS X and the only application environment for iOS. An integrated development environment called Xcode supports application development for both platforms. The combination of this development environment and Cocoa makes it easy to create a well-factored, full featured application. One can use several programming languages when developing Cocoa software, but the essential required language is Objective-C (a superset of ANSI C).

The most important Cocoa class libraries come packaged in two core frameworks for each platform: Foundation and AppKit for MAC OS X, and Foundation and UIkit for iOS. The Foundation, AppKit, and UIKit frameworks are essential to Cocoa application development, and all other frameworks are secondary and elective. Classes, functions, data types, and constants in Foundation and the AppKit have a prefix of "NS", while classes, functions, data types, and constants in UIKit have a prefix of "UI".

With creation of a project in Xcode, it sets up the initial development environment-using project templates suited to different Cocoa project types. Classes, functions, data types, and constants in Foundation and the AppKit have a prefix of "NS", while classes, functions, data types, and constants in UIKit have a prefix of "UI".

The specification of a class in Objective-C requires two distinct parts: namely the interface and the implementation. The interface is usually in a .h file and contains the class declarations as well as the class associated instance variables and methods. An implementation that actually defines a class is usually in a .m file and contains the actual code for the class methods. A class in Objective C can declare not only class methods but also instance methods. A class method is a method that performs some operation on the class itself. An instance method is a method whose execution is scoped to a particular instance of the class. The leading (+) sign tells the Objective-C compiler that the method is a class method. On the other hand, a leading dash (-) sign indicates an instance method.

Model View Controller (MVC) defines the overall architecture of the Cocoa frameworks. It is a

high level pattern for organizing large groups of cooperating objects into distinct subsystems: the Model, the View, and the Controller. The Application Kit contains objects for use in both the View and Controller subsystems.

The concept of the view controller is briefly introduced next. The job of a view controller is to manage a simple screen from your application. The three files together (the .xib, .h, and .m) form the implementation of a view controller. The MainMenu.xib constitutes the layout that is displayed on the screen while in the application. Once all the buttons and labels were created then a different screen for the help menu was created. A summary of different buttons, screens and labels is given as follows

- Buttons
  - oSetting 1
  - oSetting 2
  - oSetting 3
  - oHelp
  - oStop All
  - oOpen Connection
  - oClose Connection
- Screens
  - OHelp
- Labels
  - oPress Help
  - oCurrent Setting
  - oTerrain Sensing Vehicle

Figure 1. GUI for ATSV

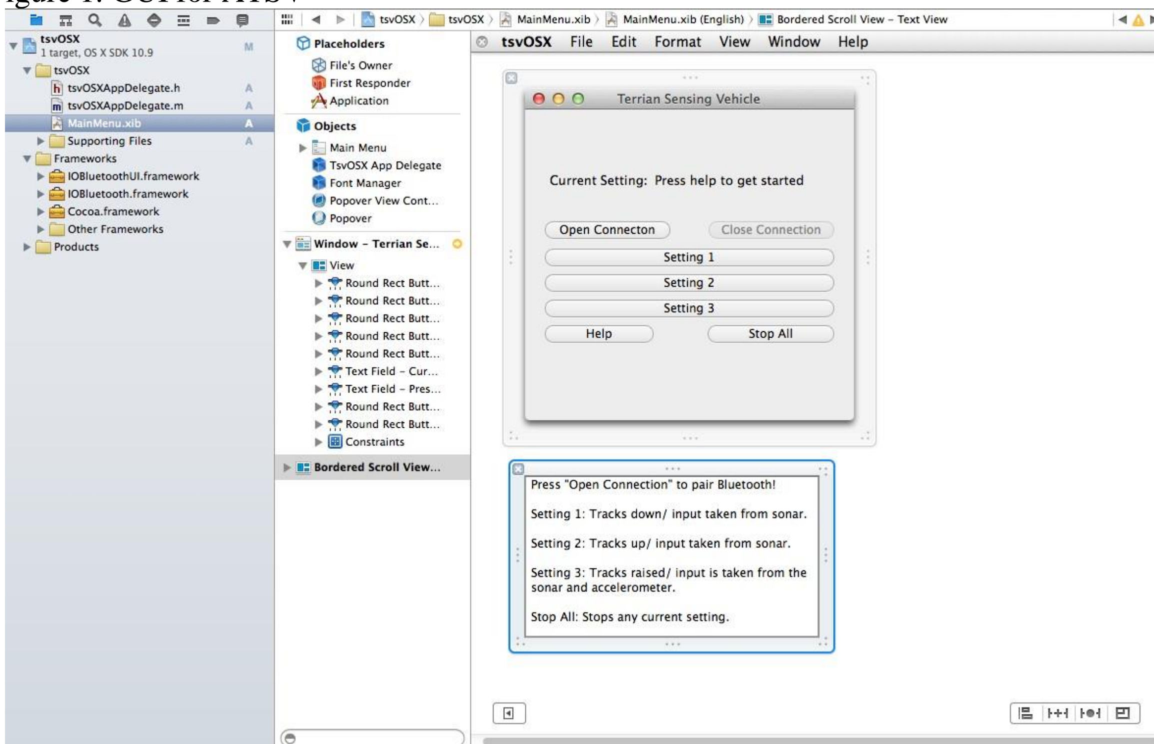

Figure 2 below depicts the system software block diagram.

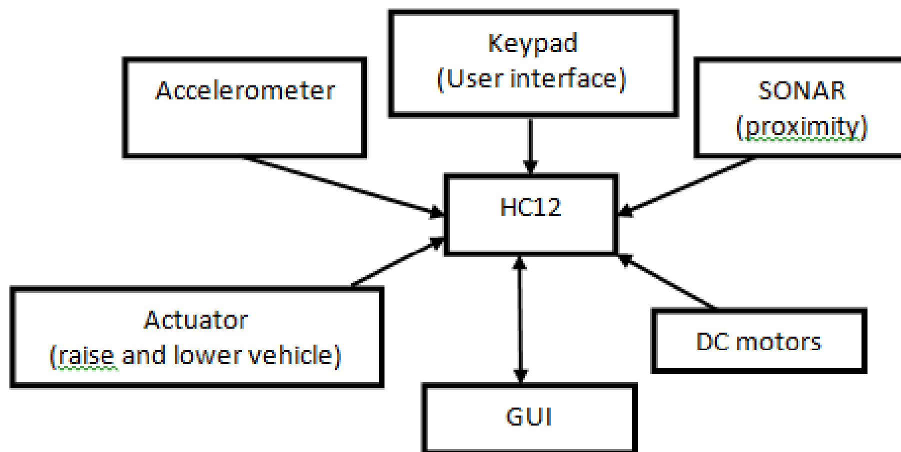

Figure 2. Terrain Sensing Vehicle System - Software Block Diagram

The ATSV Application Software Flow Chart shown in Figure 3 nicely describes how the Bluetooth application is setup.

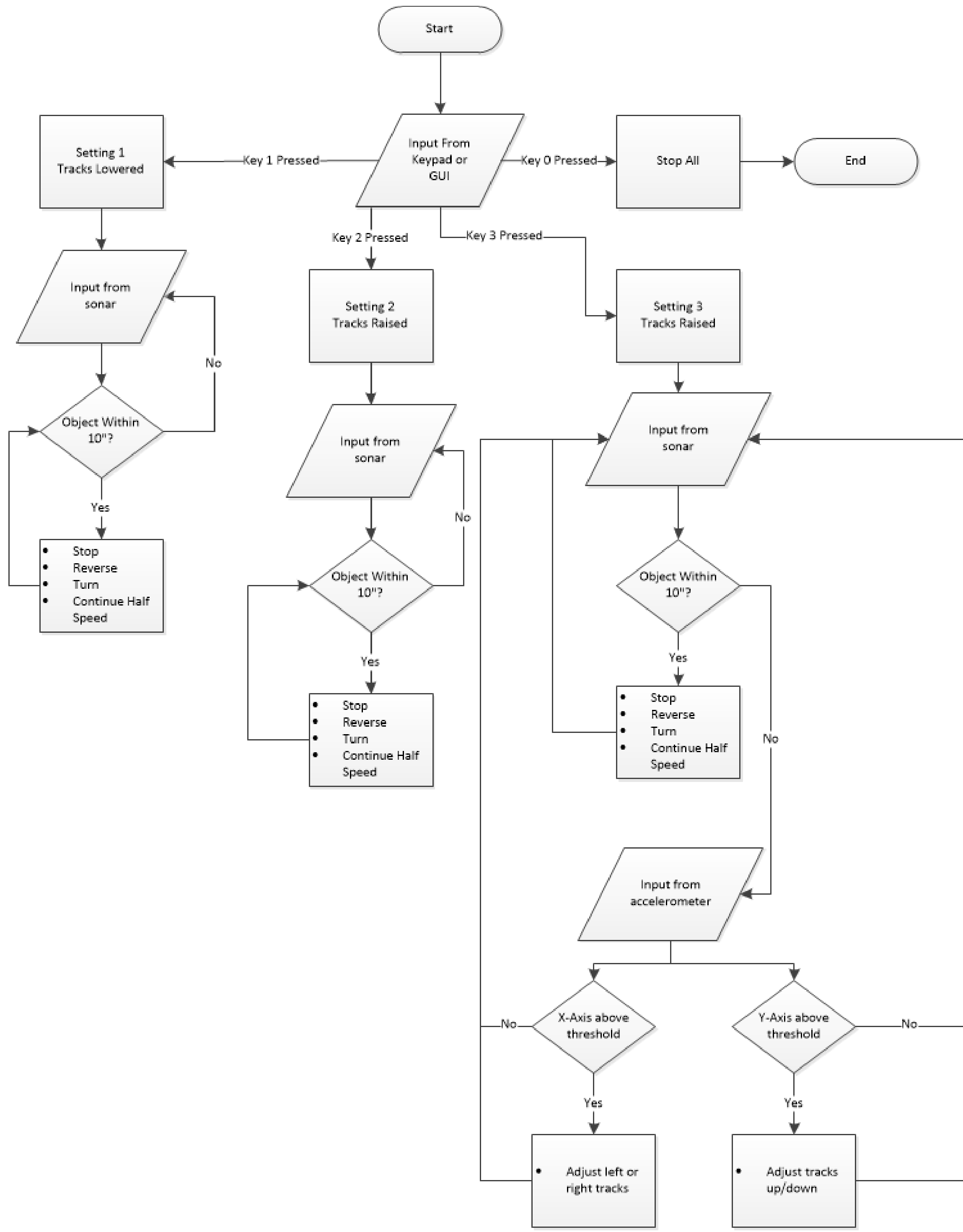

Figure 3. ATSV Flow Chart

The Bluetooth framework supports user-space access to Bluetooth devices, including both C and Objective-C APIs. Bluetooth is not a wireless networking solution, such as AirPort; rather, it is an alternative to the IrDA (Infrared Data Association) standard. Bluetooth devices can communicate at ranges of up to 10 meters and do not need to be in direct sight of each other.

The foundation of OS X Bluetooth support is Apple's implementation of the Bluetooth protocol stack. Figure 4 below shows the Bluetooth protocol stack that's built into OS X version 10.2 and later.

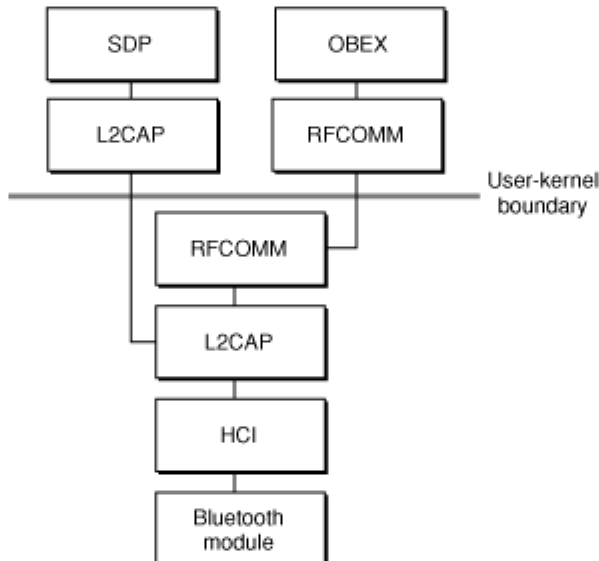

Figure 4. Bluetooth Protocol Stack

For this project the following #import directives similar to #include directives were used:

```
#import <Cocoa/Cocoa.h>
#import <IOBluetooth/IOBluetooth.h>
#import <IOBluetooth/objc/IOBluetoothSDPUUID.h>
#import <IOBluetooth/objc/IOBluetoothRFCOMMChannel.h>
#import

<IOBluetoothUI/objc/IOBluetoothDeviceSelectorController.h>
```

A brief account of a different class used in the project is depicted below.

Shown below is a screen shot of the integrated Mac Bluetooth client. This is the screen one sees when the Open Connection button is pressed. In this case, the Bluetooth chip on the development board is named "Test". To pair, just select Test and the pairing process will begin. Once paired, the window will go away and will grey out the Open Connection Button.

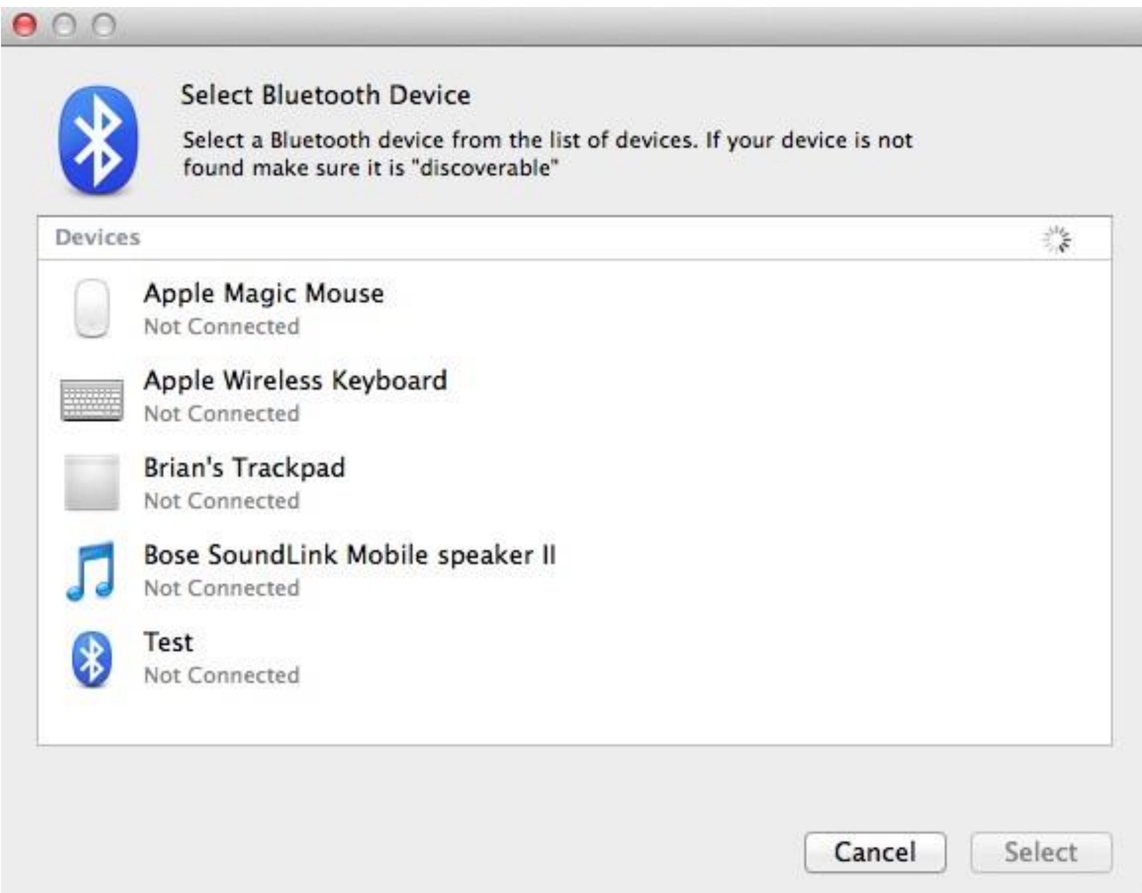

Figure 5. Integrated Mac Bluetooth Client

Here are the methods to close the RFCOMM connection channel and close the device connection on the device. You will notice the code under the closeDeviceConnectionOnDevice method contains the information to re-enable the open connection button and disable the close button to signify that there is no current Bluetooth connection.

```
- (void)closeRFCOMMConnectionOnChannel:(IOBluetoothRFCOMMChannel*)channel
{
        if ( mRFCOMMChannel == channel )
        {
          [mRFCOMMChannel closeChannel];
        }
}

- (void)closeDeviceConnectionOnDevice:(IOBluetoothDevice*)device
{
        if ( mBluetoothDevice == device )
        {
          IOReturn error = [mBluetoothDevice closeConnection];
```

```
            if ( error != kIOReturnSuccess )
            {
                        // Failure to close the connection, maybe the device is busy, as soon as the
device is not busy anymore the connection closes itself.
                        NSLog(@"Error - failed to close the device connection with error
%08x.\n", (unsigned int)error);
            }

            [mBluetoothDevice release];
            mBluetoothDevice = nil;
        }

        // Re-enable the open button so the user can restart the
        sequence: [mOpenButton setEnabled:TRUE];

        // Since we are closed also disables the close
        button: [mCloseButton setEnabled:FALSE];
}
```

The following are all of the methods that are called when “things” happen on the Bluetooth connection. The rfcommChannelOpenComplete method is there to enable the close button once a successful Bluetooth connection is established. Following the open complete method is the method that took the longest time to get working. This method is required to tell the RFCOMM channel which data to send and how long the data is expected to be. In this case one only needs to send one number to the development board so the code was mRFCOMMChannel writeSync:"1" length:1. This conforms to the method by writing “1” to the current RFCOMM channel and the data length is 1 since it only sends one byte to the development board. The last method is invoked when the Bluetooth device loses connection to the Bluetooth device on the development board; this method simply closes the device connections and re-enables the open connection button.

```
// Called by the RFCOMM channel on us once the baseband and rfcomm connection
is completed:
- (void)rfcommChannelOpenComplete:(IOBluetoothRFCOMMChannel*)rfcommChannel
status:(IOReturn)error
{
        if ( error != kIOReturnSuccess )
        {
            NSLog(@"Error - failed to open the RFCOMM
            channel"); [self
            rfcommChannelClosed:rfcommChannel];
            return;
        }
    // Now that the channel is successfully open we enable the close
        button: [mCloseButton setEnabled:TRUE];
}
```

```
// Called by the RFCOMM channel on us when new data is received from the channel:
 - (void)rfcommChannelData:(IOBluetoothRFCOMMChannel *)rfcommChannel data:(void
 *)dataPointer length:(size_t)dataLength
 {
        unsigned char *dataAsBytes = (unsigned char *)dataPointer;

        while ( dataLength-- )
        {
          [self addThisByteToTheLogs:*dataAsBytes];
          dataAsBytes++;
        }
 }

 // Called by the RFCOMM channel on us when something happens and the connection is lost:
 - (void)rfcommChannelClosed:(IOBluetoothRFCOMMChannel *)rfcommChannel
 {
        // waits for a second and closes the device connection :
        [self performSelector:@selector(closeDeviceConnectionOnDevice:)
 withObject:mBluetoothDevice afterDelay:1.0];
 }
```

Now that all of the code is almost complete the task to link the buttons to the appropriate buttons in the MainMenu.xib is accomplished. Xcode Assistant Editor in Xcode is accessed by simple two-finger click on the button. Simply click the bubbles next to the Sent Actions and reference the outlets dragging them to the correct method for labels and actions as depicted below:

This project was successfully designed and tested by the student who is also the coauthor of this paper. Integrating the Bluetooth technology with MacBook Pro, connecting with the Autonomous Terrain Sensing Vehicle, and controlling the vehicle from a remote location, the user is given a fascinating experience by being able to change settings on the fly. The main point is that every great iOS iPhone or Mac app starts with a brilliant idea. Translating the idea into action requires planning. In the co-author's own words, "Before you can write any code, you have to take the time to explore the possible techniques and technologies."

The core infrastructure of a Mac or an iOS app is built from objects in the UIKit or an AppKit framework. There are some resources that must be present in all Mac-based apps. Most apps include images, sounds, and other types of resources for presenting the app's content, but the App store may also require some specific resources. You may like to refer to Mac Developer Library and Programming Guide[1] for further details. With this project, the authors wish to share this knowledge with other people that may be having issues with Bluetooth integration with Apple devices and development boards.

Lessons Learned

A few lessons have been learned that will help in the successful delivery of this laboratory project module.

- Instructors for the capstone senior project must be comfortable in working with a wide variety of students at different cognitive levels.
- Instructors should be comfortable in working with young, energetic students. They must be flexible and be willing to take time out from the scheduled lesson plan to fill in gaps in student knowledge.
- It is important to keep the activities exciting and varied when teaching the program module.
- For success with this activity the student will need to learn how to develop the software so the vehicle can move with no problems.
- The students will learn how to code for the different vehicle settings.
- All of the settings have to be programmed and coded to work together.

Concluding Remarks

In conclusion, it can be stated that with proper guidance, monitoring and diligent care, the technology students can be exposed earlier to Xcode, Cocoa framework, Objective-C design pattern, Bluetooth development platform. This will go a long way in motivating them,[1] eliminating their fear, improving their understanding and enhancing their quality of education. Highly recommend this approach to attracting and retaining students to the embedded computing, and wireless networking.

All developed source code and curriculum material is available for use. Besides the author, the student coauthor is also very positive about the laboratory outcome and appreciates the enhanced understanding of wireless sensor networks and wireless API for Bluetooth and Zigbee applications. With proper mentoring, capable tutelage, and guidance, these burgeoning and talented young students will contribute to the best practices in implementing future smart phone applications development.

Bibliography

1. Mac Developer Library, Apple Incorporated, October 2013, http://developer.apple.com
2. Cleave, D.A., *Terrain Sensing for Unmanned Vehicles avoids Rocky Roads*, October 2008, The MITRE Digest, http://www.mitre.org/news/digest/advanced_research/10_08/sensing.html.
3. Muqri, M., Shakib, J., *A Taste of Java-Discrete and Fast Fourier Transforms,* American Society for Engineering Education, AC 2011-451.
4. Shakib, J., Muqri, M., *Leveraging the Power of Java in the Enterprise*, American Society for Engineering Education, AC 2010-1701.
5. Learning Objective-C: A Primer, iOS Developer Library, http://developer.apple.com/devcenter/ios/gettingstarted/docs/objectivecprimer.actio n

4. The Objective- C Programming Language, February 2003, http://pj.freefaculty.org/ps905/ObjC.pdf

5. Altenberg, B., Clarke, A., Mougin, P., *Become an Xcoder : Start Programming the Mac Using Objective-C,* CocoaLab, 2008, http://www.e-booksdirectory.com/details.php?ebook=3832
6. Kochan, S. G., *Programming in Objective-C*, Addison-Wesley, August 2011.
7. Cocoa Developers Guide, Apple Developer Publications, December 2010, http://itunes.apple.com/us/book/cocoa-fundamentals-guide/id409921412?mt=11
8. Deitel, H.M., Deitel, P.J., *Java How to program*, Prentice Hall, 2003

Appendix A: Bluetooth Project Class Definitions and Pairing Methods

IOBluetooth
An instance of IOBluetoothDevice represents a single remote Bluetooth device.
An IOBluetoothDevice object may exist independent of the existence of a baseband connection with the target device. Using this object, a client can request creation and destruction of baseband connections, and request the opening of L2CAP and RFCOMM channels on the remote device. Many of the other APIs in the IOBluetooth framework will return this object, or its C counterpart (IOBluetoothDeviceRef).

IOBluetoothSDPUUID
An NSData subclass that represents a UUID as defined in the Bluetooth SDP spec.
The IOBluetoothSDPUUID class can represent a UUID of any valid size (16, 32 or 128 bits). It provides the ability to compare two UUIDs no matter what their size as well as the ability to promote the size of a UUID to a larger one.

IOBluetoothRFCOMMChannel

An instance of this class represents an rfcomm channel as defined by the Bluetooth SDP spec. An RFCOMM channel object can be obtained by opening an rfcomm channel in a device, or by requesting a notification when a channel is created (this is commonly used to provide services).

IOBluetoothDeviceSelectorController
A NSWindowController subclass to display a window to initiate pairing to other Bluetooth devices. Implementation of a window controller to return a NSArray of selected Bluetooth devices. This class will handle connecting to the Bluetooth Daemon for the purposes of searches, and displaying the results. This controller will return an NSArray of IOBluetoothDevice objects to the user.

In Xcode environment, the object variable (which is the IBOutlet) and the action (IBAction) were created.

```
{
   IBOutlet id
   mCloseButton; IBOutlet
   id mOpenButton;

         // Bluetooth variables: IOBluetoothDevice
         *mBluetoothDevice; IOBluetoothRFCOMMChannel
         *mRFCOMMChannel;
}
@property (assign) IBOutlet NSTextField *myText;
@property (assign) IBOutlet NSWindow *window;
@property (assign) IBOutlet NSPopover *help;
@property (assign) IBOutlet NSButton *setting1;
@property (assign) IBOutlet NSButton *setting2;
@property (assign) IBOutlet NSButton *setting3;
```

```
@property (assign) IBOutlet NSButton *stop;


- (IBAction)setting1:(id)sender;
- (IBAction)setting2:(id)sender;
- (IBAction)setting3:(id)sender;
- (IBAction)stop:(id)sender;
- (IBAction)help:(id)sender;
- (IBAction)pair:(id)sender;
- (IBAction)unpair:(id)sender;
```

Each button was assigned to an IBOutlet, which are macros defined to denote variables and methods that can be referred to in view controller. IBAction resolved to “void” and IBOutlet resolves to nothing, but they signify to Xcode and view controller that these variables and methods are being used in view controller to link UI elements to the code. When defining each button and view, for ease of use it is a good practice to use common names so one can remember the buttons and labels easily.

```
// Start of Bluetooth pairing


- (IBAction)closeConnectonAction:(id)sender;
- (IBAction)openConnectionAction:(id)sender;
```

Create a BOOL so if the user closes the window it will also close the Bluetooth connection:

```
// Delegate on window to know when the user closes the window
// so we can close everything.
- (BOOL)windowShouldClose:(id)sender;
```

Create methods to handle the RFCOMM connection:

```
// Methods to handle the Baseband and RFCOMM connection:
- (BOOL)openSerialPortProfile;
- (void)closeRFCOMMConnectionOnChannel:(IOBluetoothRFCOMMChannel*)channel;
- (void)closeDeviceConnectionOnDevice:(IOBluetoothDevice*)device;
```

The following methods are called when “things” happen on the Bluetooth connection:

```
// Called by the RFCOMM channel on us once the baseband and rfcomm connection is
completed:
- (void)rfcommChannelOpenComplete:(IOBluetoothRFCOMMChannel*)rfcommChannel
status:(IOReturn)error;

// Called by the RFCOMM channel on us when new data is received from the channel:
- (void)rfcommChannelData:(IOBluetoothRFCOMMChannel *)rfcommChannel data:(void
*)dataPointer length:(size_t)dataLength;
```

```
// Called by the RFCOMM channel on us when something happens and the connection is lost:
- (void)rfcommChannelClosed:(IOBluetoothRFCOMMChannel *)rfcommChannel;
```

The .m file is the main part of the code, where one enters all of the code and executes the task when the method is invoked. The obvious step is to import the .h file into the .m file so that the methods are in context. It is noticed that when one looks at the .h file they have the same methods as in .m file because these are automatically created, they just need to added to your code.

Import .h file:

```
#import "tsvOSXAppDelegate.h"
```

Adding the @synthesize, creates a getter and setter method for each of the variables:

```
@synthesize window, help, setting1, setting2, setting3, stop, myText;
```

IBActions code segment is added. Notice that the [mRFCOMMChannel writeSync “(number)” length:1] will write a 0, 1, 2 or 3 to the Bluetooth channel once the channel is connected and the correct button is pressed. These correspond to getkey functions on the development board so when the development board receives a 0, 1, 2 or 3 it will start whatever method the numbers correspond to. For example, if a user presses 1, Setting 1 will start; 2, Setting 2 will start etc...When 0 is pressed it will stop all functions. Proceed to setup of the help button to display the second window to show all of the settings and what each setting does to help the end user.

```
- (IBAction)setting1:(id)sender {
   myText.stringValue = [NSString stringWithFormat:@"Setting 1"];
   [mRFCOMMChannel writeSync:"1" length:1];
}
- (IBAction)setting2:(id)sender {
   myText.stringValue = [NSString stringWithFormat:@"Setting 2"];
   [mRFCOMMChannel writeSync:"2" length:1];
}
- (IBAction)setting3:(id)sender {
   myText.stringValue = [NSString stringWithFormat:@"Setting 3"];
   [mRFCOMMChannel writeSync:"3" length:1];
}
- (IBAction)stop:(id)sender {
   myText.stringValue = [NSString stringWithFormat:@"Stop All!"];
   [mRFCOMMChannel writeSync:"0" length:1];
}
- (IBAction)help:(id)sender {
   [[self help] showRelativeToRect:[sender bounds] ofView:sender
preferredEdge:NSMaxYEdge];
```

}

Bluetooth Pairing Methods
The Open Connection and Close Connection button corresponds to pair and unpair method. The pair method opens the SerialPortProfile, which is integrated with the Mac and opens the Bluetooth window to activate search and pair once the Bluetooth device is plugged into the development board. Also, when the pair button is pressed it will grey out to prevent pressing it twice, this feature is beneficial whether the device is paired or not.

```
-(IBAction)pair:(id)sender{
    if ( [self openSerialPortProfile] )
    {
        [mOpenButton setEnabled:FALSE];
    }
}
-(IBAction)unpair:(id)sender{
  // The button activation opens a new connection, hence not need to re-enable it:
        [mCloseButton setEnabled:FALSE];

        // Accomplish the real work of closing the connection:
        [self closeRFCOMMConnectionOnChannel:mRFCOMMChannel];
}
```

This method is executed when the user closes the window. Checks to see if mRFCOMMChannel or mBluetoothDevice is not nil then it will close the RFCOMMConnectionOnChannel:mRFCOMMChannel. It thereby closes the connection on the open RFCOMM channel.

```
- (BOOL)windowShouldClose:(id)sender
{
        //Check whether an open connection close:
        if ( ( mRFCOMMChannel != nil ) || ( mBluetoothDevice != nil ) )
                [self closeRFCOMMConnectionOnChannel:mRFCOMMChannel];

        return TRUE;
}
```

Bluetooth specific code is written to open the serial port profile. It opens the device selector and starts searching for nearby Bluetooth devices. This allows device and pair enable to internal Bluetooth chip. At the end of the code it is seen that RFCOMM channel must be retained. This is accomplished by assigning mBluetoothDevice to device and retaining the mBluetoothDevice and mRFCOMMChannel.

```
- (BOOL)openSerialPortProfile
{
```

```
IOBluetoothDeviceSelectorController      *deviceSelector;
    IOBluetoothSDPUUID                    *sppServiceUUID;
    NSArray                               *deviceArray;

// The device selector will provide UI to the end user to find a remote device
deviceSelector = [IOBluetoothDeviceSelectorController deviceSelector];

    if ( deviceSelector == nil )
    {
      NSLog( @"Error - unable to allocate IOBluetoothDeviceSelectorController.\n" );
      return FALSE;
    }

    // Create an IOBluetoothSDPUUID object for the chat service UUID
    sppServiceUUID = [IOBluetoothSDPUUID
uuid16:kBluetoothSDPUUID16ServiceClassSerialPort];

    // Tell the device selector what service we are interested in.
    // It will only allow the user to select devices that have that service.
    [deviceSelector addAllowedUUID:sppServiceUUID];

    // Run the device selector modal.  This won't return until the user has selected a device and
the device has
    // been validated to contain the specified service or the user has hit the cancel button.
    if ( [deviceSelector runModal] != kIOBluetoothUISuccess )
    {
      NSLog( @"User has cancelled the device selection.\n" );
      return FALSE;
    }

    // Get the list of devices the user has selected.
    // By default, only one device is allowed to be selected.
    deviceArray = [deviceSelector getResults];

    if ( ( deviceArray == nil ) || ( [deviceArray count] == 0 ) )
    {
      NSLog( @"Error - no selected device.  ***This should never happen.***\n" );
      return FALSE;
    }

    // The device to be accessed is addressed via the first array
    // (even if the user somehow selected more than one device
    IOBluetoothDevice *device = [deviceArray objectAtIndex:0];


    // Finds the service record that describes the service (UUID) being sought
    IOBluetoothSDPServiceRecord *sppServiceRecord =
```

```
                    [device getServiceRecordForUUID:sppServiceUUID];

        if ( sppServiceRecord == nil )
        {
          NSLog( @"Error - No spp service in selected device.  ***This should never happen
          since the selector forces the user to select only devices with spp.***\n" );
          return FALSE;
        }
        // Require to connect a device and an RFCOMM channel ID opens on the device
          UInt8   rfcommChannelID;
        if ( [sppServiceRecord getRFCOMMChannelID:&rfcommChannelID] !
                                                    = kIOReturnSuccess )
        {
          NSLog( @"Error - no spp service in selected device.  ***This should never happen an
           spp service must have an rfcomm channel id.***\n" );
          return FALSE;
        }
        // Open asynchronous  rfcomm channel once the open sequence completed
        // invokes  implementation of "rfcommChannelOpenComplete:"
        if ( ( [device openRFCOMMChannelAsync:&mRFCOMMChannel
        withChannelID:rfcommChannelID delegate:self] != kIOReturnSuccess )
&& ( mRFCOMMChannel != nil ) )
        {
          // Something went bad; look at the error
          // If the device connection is left open close it and return an
          error: NSLog( @"Error - open sequence failed.***\n" );

          [self closeDeviceConnectionOnDevice:device];

          return FALSE;
        }

        // So far a lot of stuff went well, so we can assume that the device is a good one and
        //that rfcomm channel open process is going well. So we keep track of the device and
        // we (MUST) retain the RFCOMM channel:
        mBluetoothDevice = device;
        [mBluetoothDevice  retain];
        [mRFCOMMChannel
        retain];

        return TRUE;
}
```