Remote Display For Smartphone

A graduate project submitted in partial fulfillment of the requirements
For the degree of Master of Science
in Computer Science

By

Peter Chyla

May 2013
The thesis of Peter Chyla is approved:

__________________________  ___________________________
Richard Covington, Ph.D.     Date

__________________________  ___________________________
Michael Barnes, Ph.D.         Date

__________________________  ___________________________
Jeff Wiegley, Ph.D., Chair   Date
Table of Contents

Signature Page .................................................................................................................. ii
Abstract .............................................................................................................................. v
   Introduction ..................................................................................................................... 1
   Similar Work .................................................................................................................. 2
   Project Goals: ............................................................................................................... 2
End Product Requirements .............................................................................................. 4
   API Requirements: ...................................................................................................... 4
   Hardware Prototype Requirements: ............................................................................. 4
   Demo Application Requirements: ............................................................................... 5
High level API design: ...................................................................................................... 6
   API Must Function Transparently With a Wireless Communication Link ................. 6
   API Must Allow an Application to Draw Any Shape On the Remote Display .......... 7
   API Must Provide Mechanism for Shapes to be Scrolled and Zoomed .................... 7
   API Must Have Mechanism to Handle Remote User Input ...................................... 8
   API Must Allow an Application to Write Text on the Remote Display .................... 9
RemoteDisplayAPI Javadoc ........................................................................................... 10
RemoteDisplayObject JavaDoc ....................................................................................... 13
Hardware Prototype design ............................................................................................ 22
   Wireless communication with Smart Phone .............................................................. 23
   Color display 128x128 or greater ................................................................................. 23
   User interface which will allow scrolling, zooming, and other user operations .......... 23
   Low cost ....................................................................................................................... 23
   Low power consumption ............................................................................................. 23
   Small and Lightweight ................................................................................................. 23
Demo Application Design: ............................................................................................ 25
   Map and GPS display ................................................................................................. 25
   Text message display .................................................................................................. 25
   Text input mode ......................................................................................................... 26
API implementation: ................................................................. 27
Protocol Implementation: .......................................................... 27
Firmware Implementation: .......................................................... 28
  Bluetooth communication implementation .................................. 30
  Display Task implementation .................................................... 31
  Touch Input Implementation ...................................................... 34
Android API implementation: ...................................................... 36
Application Implementation/API testing: ........................................... 39
  Bike Map/Speed display ............................................................ 39
  Text Message Display ............................................................. 41
  Text input mode ...................................................................... 42
Application Performance Analysis: ................................................. 43
  Application/Firmware Timing ..................................................... 43
  Power consumption of remote display device ............................... 45
Tools used .................................................................................. 47
  Embedded Workbench ............................................................. 47
  Eclipse ..................................................................................... 47
  Phone Simulator ...................................................................... 48
  Samsung Galaxy S .................................................................... 48
Conclusion ................................................................................ 49
  Lessons Learned ...................................................................... 49
  Further Work .......................................................................... 49
Bibliography ................................................................................ 50
Abstract

Remote Display for Smartphone

By

Peter Chyla

Master of Science in Computer Science

Modern smartphones are very capable devices. Thanks to the ease of writing and using applications, the cell phone has morphed from a device which allows its user to communicate with others into an internet connected, multi-purpose gadget. Smartphone applications exist for almost everything imaginable, but in a few cases, such as skiing or cycling, physically accessing the smartphone to use an application is not practical.

As a solution to the above stated problem, this thesis presents the design and implementation of a prototype “Remote Display for Smartphone” device and associated Android Library. The “Remote Display for Smartphone” is a small embedded device which features an LCD display and wireless connectivity to a smartphone. Android applications can use this device to display data and to receive input from the user without the need for the user to physically access the phone.

This project used an “ARM Cortex-M3 MINI STM32” Development Board with a 2.4” TFT LCD Touchscreen as a hardware prototype for the Remote Display for Smartphone. An RN42 Bluetooth module was added to the development board to allow wireless connectivity to the smartphone. The firmware for the Remote Display for Smartphone was developed in C using the IAR Systems, Embedded Workbench for ARM. The “Remote Display Device” Android library was developed using the JAVA programming language and the Eclipse based Android SDK.

As a demonstration of the capabilities of the Remote Display for Smartphone a sample Bike Map application which utilizes the Remote Display API was developed.
Introduction
Modern smartphones are very capable devices. Hundreds of thousands of applications, which allow users to do anything from playing simple games to depositing a check, are available for download. The smartphone is slowly eating away the market for all other gadgets (Leber). Gone are the days of stand-alone mp3 players. Want to listen to some music on the go? Use your smartphone. Need to get directions to the concert you are attending? Forget that old clunky GPS you once had attached with a suction cup to your windshield, and use your smartphone. Unlike most stand-alone gadgets, all smartphones have internet access and provide a platform for developers to make applications to suit any need without any additional hardware.

For the scope of this paper, the hundreds of mountain biking applications available in Google’s Android market place are of interest. Multiple applications exist which allow riders to track their riding. The applications log rides and allow users to see the speed, elevation and distance covered. Other applications provide trail maps to assist with navigation and to allow riders to locate new trails. During the past year or so, the popularity of one such mountain biking application, Strava, has exploded. Strava allows riders to log their time on various segments of a trail and to automatically update their rides on an online profile. Using this application it is easy to see if that new training program has increased a rider’s performance and to brag about how many miles a rider covered that day.

Despite of the overwhelming positive attributes of modern smart phones, all of the mountain biking applications available for smart phones share one major drawback: accessing a smartphone during a mountain bike ride is difficult. New smart phones such as the Samsung Galaxy S3 cost as much as $899 dollars (Amazon.com) when not subsidized by a carrier, and although though cases exist, mounting a smart phone to a mountain bike while riding a narrow rocky trail is just asking for a broken phone.

As an alternative to mounting an expensive phone to a mountain bike, one can acquire a standalone GPS device such as the Garmin Edge 810. However, such Garmin Edge 810 devices start at $499.99 (Garmin) and, while the most recently released versions do provide smart phone connectivity via Bluetooth they do not allow for third party applications and only make very limited use of their connectivity.

In addition to a large profit margin, one of the reasons for the high cost of the Garmin Edge device is the fact that it is a stand-alone GPS device, which requires a powerful processor and a large battery to provide reasonable battery life. All modern smartphones already have the GPS hardware and large battery built in, so instead of providing duplicate hardware functionality, this paper discusses the creation of an API which will allow smartphone applications to use a small, inexpensive remote display for communicating information with the user while the smartphone is safely tucked away. A
demo Andorid mountain bike application which uses the API discussed in this paper displays a trail map along with the user’s location, current speed and provides access to text messages through the remote display.

The Remote Display for Smartphone relies on a number of modern products and technologies to function. For example, a few years ago, phones did not have the capability to execute custom programs. Their firmware was loaded during production, and whatever programs were installed by the phone’s manufacturer were the only programs that could be executed on that device. The Remote Display for Smartphone relies on this ability of smartphones to execute third party applications. Furthermore, the ease of application development and distribution makes it possible for individuals to develop and distribute applications to a wide audience, enabling potential developers to use the Remote Display for Smartphone in their applications. The Remote Display for Smartphone also relies on the fact that Bluetooth is commonplace on modern smartphone and that Bluetooth makes separate physical bodies appear connected in a logical sense.

**Similar Work**
The Pebble Smart Watch is a device which as of May 2013 is accepting pre-orders for $150. The Pebble Smart Watch is quite similar to the “Remote Display For Smart Phone” discussed in this paper. Just like the Remote Display For Smartphone, the pebble smartwatch connects to a smartphone via Bluetooth and it allows for data to be displayed to the user, and for the user to control some aspects of the smartphone without the need for physically accessing the phone. The major difference between the Pebble SmartWatch and the Remote Display for Smartphone proposed in this paper is that the Pebble SmartWatch is smart, while the Remote Display for Smartphone is not. In comparison to the Pebble SmartWatch, the Remote Display for Smartphone is a very simple device. The pebble smartWatch runs applications, while the Remote For Smartphone simply acts as an I/O device for applications running on the smartphone. The approach taken by the Pebble SmartWatch allows for more flexibility at the cost of higher complexity. In order to add a feature to the Pebble Smartwatch, an application must be developed for both the Smartphone and the smartwatch, complicating development and increasing future maintenance costs for application developers. With the Remote for Smartphone, any Android application can choose to use the remote display device without the need to write any separate code.

**Project Goals:**
1. Design simple object oriented API for communicating with remote display device.
2. Create hardware prototype of remote display device.
3. Implement the API on both the smartphone and remote display device.
4. Implement a sample application to demonstrate API and hardware device operation.
**END PRODUCT REQUIREMENTS**

The final result of this project is an Android API which allows Android applications to communicate with a prototype hardware implementation of the “Remote for Smartphone” and an application to demonstrate the capabilities of the API.

**API Requirements:**

**Purpose of API:**

Provide an easy to use interface for application developers to communicate with and interact with the remote display. The “API” includes three separate parts: Android interface, communication protocol and remote display device firmware.

1. The API must function transparently over a wireless communication link.
   a. A wireless connection between the Remote Display and the smartphone is necessary because the remote display is intended to be used in situations where a wired connection is not possible. For example, the Remote Display may be attached to a bicycle, while the smartphone may be tucked away in a backpack.

2. The API must allow an application to draw any shape on the remote display.
   a. The ability to draw any shape on the Remote Display gives applications using the Remote Display maximum flexibility and increases the number of possible use cases for the Remote Display.

3. The API must provide a mechanism for shapes to be scrolled and zoomed.
   a. The remote display device uses a small screen, so scrolling and zooming are necessary to allow complex data to be displayed in a usable manner.

4. The API must have a mechanism to handle remote user input.
   a. The remote display is intended to be used in situations where physical access to the smartphone is not practical, so providing a way for users to interact with the smartphone through the remote display increases the number of possible use cases for the Remote Display.

5. The API must allow applications to write text on the remote display
   a. The display of text on the remote display is necessary to enable users to access smartphone features with a text based interface such as text messaging.

**Hardware Prototype Requirements:**

**Purpose:**

The purpose of the hardware prototype is to provide a remote display for the above mentioned API to be developed and tested on.

**Requirements:**
1. Wireless communication with Smart Phone
2. Color display 128x128 or greater
3. User interface which will allow scrolling, zooming, and other user operations.
4. Low cost
5. Low power consumption
6. Small and Lightweight

**Demo Application Requirements:**

Purpose of Demo application:

The main purpose of this demo application is to demonstrate the capabilities and operation of the API. This application, “Bike Map” will use the RemoteForSmartPhone API and the remote display to implement a simple bike computer and navigation device.

The “Bike Map” application displays a map, the user’s location and the user’s speed on the Remote Display device. Additionally, the Bike Map application allows the user to read and write text messages using the Remote Display.

1. Read map information from properly formatted .GPX files
2. Display the above map on the remote display.
   a. The user should be able to scroll the map.
   b. The user should be able to zoom in and out of the map.
3. Display the current location of the cell phone on the remote display.
4. Display current speed of the cell phone on the remote display.
5. Provide the user the ability to send and receive text messages on the remote display.
**HIGH LEVEL API DESIGN:**

![Diagram showing system design. Highlighted parts are contributed by this project.](image)

The API was designed to meet the requirements specified in the End Product Requirements section. This section of the paper discusses how each requirement was met by the design of the API and is organized by requirement.

**API Must Function Transparently With a Wireless Communication Link**

The API fulfills this requirement by providing a layer of abstraction over the communication link. The developer writing an application which utilizes the remote display API is completely separated from the communication protocol and its associated complexities. The developer simply has to open a link and proceed to control the display using the functions provided. Under the hood, things get a little complicated. The application programmer uses JAVA to write an Android application which directly controls the remote display which is running firmware written in C. In order to implement manageable communication between the two sides of the implementation, a communication protocol was established. The data from each remote display object on the android side is copied into a buffer which resembles a C structure. This C structure is than sent over the wireless link to the remote display device where the structure is decoded and displayed as needed.
API Must Allow an Application to Draw Any Shape On the Remote Display

Two separate designs were considered when deciding how image data should be communicated and displayed on the remote display device: bitmaps and very simple vector graphics. As discussed below, each method has several advantages and disadvantages.

Raster:

Pros:

- Good image quality and detail. Possible to display full color, high quality images.
- Trivial drawing algorithm. Color of each pixel is contained in the bitmap.

Cons:

- Huge amount of data must be transferred. 128x128 display has 16K pixels, so 16bit bit map would be 32KB
- Cannot scale without losing quality. Enlarging bitmap text on display would result in loss of quality.
- Slow drawing algorithm. If the data to be displayed changes, the entire bitmap would need to be redrawn.

Vector Graphics:

Pros:

- Image data can be very small.
- Image can be scaled at will without losing quality.
- Simple images can be drawn very efficiently.

Cons:

- Difficult to create high quality images.
- Complex drawing algorithm.

Considering the qualities of each image drawing method, a very simple form of vector graphics was chosen for the design of this API. Each shape that is to be draw contains a series of points. When such a shape is to be displayed, a line is drawn between each consecutive point. Using this method allows simple graphics to be efficiently transferred over the limited bandwidth of the wireless link.

API Must Provide Mechanism for Shapes to be Scrolled and Zoomed.
In order to allow complex data to be displayed on a small screen it is critical to have an efficient means of scrolling and enlarging the graphics being displayed. Two options were considered when deciding on how to design this feature. One option was to transfer every user input to the phone, have the phone perform the proper transformation and resubmit a new set of drawing instructions for the object. This option would have resulted in significant amounts of data constantly being transferred over the wireless link, causing unneeded power consumption and slower response time. The other option considered was to have the remote display device handle the scrolling and zooming of data.

<table>
<thead>
<tr>
<th>Send touch input to phone</th>
<th>Handle Scrolling and Zooming on the Remote Display Device</th>
</tr>
</thead>
</table>
| **Pros** | • Remote Display device firmware can be simpler.  
• Unlimited flexibility | • Allows for smooth scrolling and zooming. |
| **Cons** | • Difficult to provide smooth scrolling  
• Large amount of data must be transferred  
• Low performance | • Remote Display device must buffer more data than currently being displayed  
• Adds complexity to Remote Display Device firmware |

Considering the above pros and cons, the method of having the Remote Display Device handle the Scrolling and Zooming operations was chosen.

**API Must Have Mechanism to Handle Remote User Input.**

Since the Remote for Smart Phone provides a way for the remote to control the smartphone, it is necessary for the API to provide a mechanism for user input from the device. Two types of input are considered. The user can click objects on the screen, and the can swipe over objects on the screen. Various schemes for object clicking and swiping were considered. One possibility was to transfer the exact location of the user touch input to the phone and have the phone determine which, if any objects were clicked. This solution however would not be compatible with the scrolling and zooming technique which was adopted. The phone is not aware which part of the display is currently being shown, so along with a click coordinate, additional data detailing the zoom and offset of the display would have to be transferred in order to determine which object was clicked. The other solution which was considered was that every clickable object has an identification byte which is to be transferred to the phone when that object is clicked. In this solution, the remote display device determines if the location of the users click, or swipe falls on an object. Once the remote display device has determined which object the user input interacted with, the phone is notified of the user input and must respond accordingly by calling the callback function configured for that object on the smartphone.
API Must Allow an Application to Write Text on the Remote Display

Two design options were considered for the display of text on the remote display device. The first option was to store a font on the remote display device, and have the smartphone provide a set of commands detailing how and where to draw the font. The other option was to treat text the same as all other shapes to be draw on the display. Early in the design process, the option to provide special treatment for text was adopted, however upon further analysis it was decided that providing this special treatment to text was unneeded and would add complexity to the design. In the final implementation, the Remote Display Device does not know whether it is displaying an image, or text.
RemoteDisplayAPI Javadoc

csun.Peter.Chyla.RemoteDisplayAPI
Class RemoteDisplayAPI
java.lang.Object

public class RemoteDisplayAPI
extends java.lang.Object

The RemoteDisplayAPI provides an interface to the RemoteDisplay. Objects are added to the display using the AddRemoteDispalyObject function, and uploaded to the RemoteDisplay using the UpdateRemoteDisplay function.

Author:
Peter Chyla

Constructor Summary

RemoteDisplayAPI()
Creates an instance of the RemoteDisplayAPI and establishes a Bluetooth connection with the RemoteDisplay.

Method Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Method Name</th>
<th>Method Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>AddRemoteDisplayOjbect</td>
<td>(RemoteDisplayObject o)</td>
<td>Adds RemoteDisplayObject o to the collection of RemoteDisplayObjects.</td>
</tr>
<tr>
<td>Void</td>
<td>close</td>
<td></td>
<td>Closes the connection with the remote display.</td>
</tr>
<tr>
<td>RemoteDisplayObject</td>
<td>GetRemoteDisplayObject</td>
<td>(int index)</td>
<td></td>
</tr>
<tr>
<td>Void</td>
<td><strong>UpdateRemoteDisplay()</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synchronizes the current list of RemoteDisplayObjects with the RemoteDisplay.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Methods inherited from class java.lang.Object**

- equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

**Constructor Detail**

- **RemoteDisplayAPI**
  ```java
  public RemoteDisplayAPI()
  ```
  Creates an instance of the RemoteDisplayAPI and establishes a Bluetooth connection with the RemoteDisplay.

**Method Detail**

- **close**
  ```java
  public void close()
  ```
  Closes the connection with the remote display. This function must be called whenever any activity which uses the Remote Display terminates.

- **AddRemoteDisplayObject**
  ```java
  public int AddRemoteDisplayObject(RemoteDisplayObject o)
  ```
  Adds RemoteDisplayObject o to the collection of RemoteDisplayObjects. All objects in this collection get transferred to the remote display device when the UpdateRemoteDisplay() function is called.

  **Parameters:**
  
  o - Object to be added to RemoteDisplay.

  **Returns:**
  
  The index of the added object.
GetRemoteDisplayObject

public RemoteDisplayObject GetRemoteDisplayObject(int index)

Parameters:
index -

Returns:
Returns a RemoteDisplayObject located at index. NULL if the index is not valid.

UpdateRemoteDisplay

public void UpdateRemoteDisplay()

Synchronizes the current list of RemoteDisplayObjects with the RemoteDisplay. This function should be called whenever changes are made to one of the RemoteDisplay objects.
public class RemoteDisplayObject
extends java.lang.Object

Instances of the RemoteDisplayObject class represent items to be drawn on the RemoteDisplay Device. The RemoteDisplayObject class is to be used as part of the RemoteDisplayAPI. Each object contains a set of properties and a set of commands which describe how the object should be drawn.

Author:
Peter Chyla

Constructor Summary

RemoteDisplayObject()  
Creates a new RemoteDisplayObject instance.

Method Summary

<table>
<thead>
<tr>
<th>Short</th>
<th>doClick(short val)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the val parameter matches the callback number, executes the callback.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.util.ArrayList&lt;java.lang.Short&gt;</th>
<th>getDrawCommands()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the ArrayList of draw commands.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int</th>
<th>getHeight()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the height of the object.</td>
</tr>
<tr>
<td>Short</td>
<td><code>getOnClick()</code></td>
</tr>
<tr>
<td>-------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td>Returns the value of the on_click property.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int</th>
<th><code>getSize()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>returns the value of the size parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short</th>
<th><code>getWidth()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the width of the object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int</th>
<th><code>getxPos()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>returns the value of the remote display objects x position parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int</th>
<th><code>getyPos()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the value of the yPos parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short</th>
<th><code>isScrollable()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the value of the scrollable parameter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short</th>
<th><code>isVisible()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>returns the value of the visible parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><code>setDrawCommands(java.util.ArrayList&lt;java.lang.Short&gt; drawCommands)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the drawCommands for the objects.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><code>setHeight(short height)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the height of the object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><code>setOnClick(myCallBack callback, int callbacknum)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the on Click property of the object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><code>setScalable(short scalable)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the value of the scalable parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><code>setScrollable(short scrollable)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the value of the scrollable parameter.</td>
</tr>
<tr>
<td>Void</td>
<td><strong>setSize</strong>(short size)</td>
</tr>
<tr>
<td>------</td>
<td>------------------------</td>
</tr>
<tr>
<td></td>
<td>Sets the value of the size parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><strong>setVisible</strong>(short visible)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Makes the object visible on the remote display.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><strong>setWidth</strong>(short width)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the width of the object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><strong>setxPos</strong>(short xPos)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the value of the xPos parameter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Void</th>
<th><strong>setyPos</strong>(short yPos)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the yPos parameter.</td>
</tr>
</tbody>
</table>

**Methods inherited from class java.lang.Object**

`equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

**Constructor Detail**

**RemoteDisplayObject**

`public RemoteDisplayObject()`

Creates a new RemoteDisplayObject instance.

**Method Detail**

**getxPos**

`public int getxPos()`

returns the value of the remote display objects x position parameter.

**Returns:**

x position parameter
setxPos

public void setxPos(short xPos)

Sets the value of the xPos parameter.

Parameters:

xPos - This parameter represents the X coordinate of the lower left corner of this object. If the object does not have the scrollable parameter set, this x position refers to the number of pixels from the bottom left corner of the LCD screen. If the object is scrollable, this position is appropriately scaled and added to the world coordinate system.

setSize

public void setSize(short size)

Sets the value of the size parameter.

Parameters:

size - the maximum number of bytes that this object can occupy on the remote display device. If this parameter is not set, the size is automatically set to the current size of the object.

getSize

public int getSize()

returns the value of the size parameter.

Returns:

The number of bytes that this object will occupy on the remote display device.

setVisible

public void setVisible(short visible)

Makes the object visible on the remote display.

Parameters:
visible - If set to 1, the object will be drawn, if 0 the object will not be drawn.

isVisible

public short isVisible()

returns the value of the visible parameter.

Returns:

1 if the object is visible, 0 if not.

setScrollable

public void setScrollable(short scrollable)

Sets the value of the scrollable parameter.

Parameters:

scrollable - If set to 1, the object is drawn in the world coordinate system, and can be scrolled. If set to 0, the object is drawn directly on the LCD and cannot be scrolled.

isScrollable

public short isScrollable()

Returns the value of the scrollable parameter

Returns:

Returns the value of the scrollable property of the object.

setScalable

public void setScalable(short scalable)

Sets the value of the scalable parameter.

Parameters:
scalable - If set to 1, the object is scaled using the world scale value. If set to 0 the object does not get scaled.

**setyPos**

public void setyPos(short yPos)

Sets the yPos parameter.

**Parameters:**

yPos - Represents the Y coordinate of the bottom left corner of the object. If the object does not have the scrollable parameter set, this x position refers to the number of pixels from the bottom left corner of the LCD screen. If the object is scroll-able, this position is appropriately scaled and added to the world coordinate system.

**getyPos**

public int getyPos()

Returns the value of the yPos parameter.

**Returns:**

The value of the yPos property of the object.

**setHeight**

public void setHeight(short height)

Sets the height of the object.

**Parameters:**

height - Represents the height of the object. This property is used in conjunction with the xPos, yPos and width to determine whether a click hit the object.

**getHeight**

public int getHeight()
Returns the height of the object.

**Returns:**
The value of the height parameter of the object.

---

**setWidth**

public void **setWidth**(short width)

Sets the width of the object.

**Parameters:**
width - Represents the width of the object. This property is used in conjunction with the xPos, yPos and height to determine whether a click hit the object.

---

**getWidth**

public short **getWidth**()

Returns the width of the object

**Returns:**
Returns the width property of the object.

---

**doClick**

public short **doClick**(short val)

If the val parameter matches the callback number, executes the callback.

**Parameters:**
val - If val matches the callback number set via the setOnClick method, the callback is executed.

**Returns:**
1 if matching callback number was found. 0 if not
setOnClick

public void setOnClick(myCallBack callback,
            int callbacknum)

Sets the on Click property of the object.

Parameters:

callback - The callBack function contained within this object will be called whenever this object is clicked on the remote display device.

callbacknum - This is the callback number associated with this object. In most cases this callback number should be unique for all objects, but if it is desired that multiple objects have the same callback routine, the same callback num may be used.

getOnClick

public short getOnClick()

Returns the value of the on_click property.

Returns:

Returns the value of the on_click property of the object.

setDrawCommands

public void setDrawCommands(java.util.ArrayList<java.lang.Short> drawCommands)

Sets the drawCommands for the objects.

Parameters:

drawCommands - The drawCommands tell the Remote Display Device how to draw this object. The draw commands are formated as follows: 1,X coordinate,Y coordinate,Color,X coordinate,Y coordinate,Color...X coordinate,Y coordinate,0 A line of the specified color is drawn between the coordinates specified until an end marker is found. If the scrollable property is set, the line coordinates must be in the world coordinate system. If the scrollable property is not set, the line coordinates must represent the pixel location on the physical display.
getDrawCommands

public java.util.ArrayList<java.lang.Short> getDrawCommands()

Returns the ArrayList of draw commands.

Returns:

Returns the list of commands used for drawing this object.
**Hardware Prototype Design**

This is a computer science paper, so this section is intentionally kept brief. The hardware prototype was chosen to meet the requirements specified in the hardware requirements section. The Strive Mini STM32 board shown in Figure 2 was selected. The board has the following features:

“2.4" TFT LCD Touchscreen
CPU: STM32F103VET6, TQFP 100 pins FLASH£º512K BYTES,
SRAM:64KBYTES
1 JTAG debug interface
1 power LED indicator (Green), 1 status LED (Blue)
1 RS232 port, Need crossover cable to talk to PC
Support 3 pin ISP
1 USB2.0 SLAVE port
1 Micro SD(TF) slot, uses SDIO
1 2.4 inch TFT (240X320 (touch screen) ), use FSMC 16 bits interface to control
1 SPI interfaced AT45DB161D(2M BYTES) serial FLASH
1 functional button
1 RTC battery socket
All unused GPIO pins are connected to external headers.”

![Figure 2: Prototype board shown during development of firmware](image-url)
**Wireless communication with Smart Phone**
Bluetooth was adopted as the wireless communication protocol for the remote display device. Bluetooth is available in the vast majority of Android smartphones and offers sufficient bandwidth and latency over the distance required for this application while maintaining low power consumption. The Strive Mini STM32 board did not have a Bluetooth antenna, so an RN42 Bluetooth module was purchased. Some soldering was required to connect the module to the pcb.

![Image of a circuit board with a Bluetooth module](image)

Figure 3: The RS-232 connector was de-soldered from the board to make room for the RN42 Bluetooth module.

**Color display 128x128 or greater**
The Strive Mini STM32 board chosen has a 240x320 TFT LCD display.

**User interface which will allow scrolling, zooming, and other user operations.**
The STM32 board comes with a resistive touchscreen, allowing the user to perform all of the required operations.

**Low cost**
The STM32 board costs $59.00, which for a hardware prototype is cheap.

**Low power consumption**
The chosen board uses a power efficient STM32 CortexM3 based MCU. However, the LCD display on the board is not very efficient. A transflective display would have provided a better image quality while using less power, but no evaluation boards with such a display were found.

**Small and Lightweight**
The Mini STM32 board is smaller than most other available boards and is fairly lightweight.


**DEMO APPLICATION DESIGN:**

The demo application is meant to demonstrate the capabilities of the Remote for SmartPhone API by implementing a simple mountain bike computer application. The demo application is implemented as an Android Activity which uses the Remote Display API for communicating with the Remote Display hardware prototype.

**Map and GPS display**

This is the default function of the application. In this mode, a series of GPX files which contain formatted map information is read in and transformed into a set of draw commands for the Remote Display device. The map information is transferred to and displayed on the Remote Display device. Additionally, the users location is displayed on the map as a red marker, and in the bottom right corner, the current speed of the user in miles per hour is displayed at all times. As discussed in the API design section, the scrolling and zooming of the map occurs directly on the remote display device. The updating of the user location and speed is triggered by a change in the users’ location as reported by the Android GPS device.

If the users’ cell phone receives a text message, a notification icon will appear on the screen allowing the user to transition to the Text Message display mode by clicking on the icon.

![Figure 4: Design drawing of the Map and GPS display.](image)

**Text message display**

In this mode, no map information will be displayed on the screen. Instead, the content of the most recently received text message will be visible. The user will have the option of closing the text message by clicking on a “close” button or of replying to the message by clicking a “reply” button.
The text input mode of the Application uses a keyboard very similar to 8pen. (8pen). The input works by dividing the screen into 5 regions. As the user moves his finger between the regions in the correct pattern, letters are added to the text.

Figure 6: Design drawing of Text input mode.
**API implementation:**
The API implementation consisted of three parts: Protocol implementation, firmware implementation and Android side implementation.

The API implementation proved to be significantly more challenging than expected. The primary difficulty was that three parts of the system were being implemented concurrently. As a new feature was added to the firmware, the protocol and the Android side of the API had to be advanced along in order to allow the new firmware feature to be tested. Thanks to the design effort before the beginning of implementation, no significant architectural changes were required after implementation got under way, but the lack of facilities to test each component of the system independently proved quite troublesome.

**Protocol Implementation:**
The communication protocol between the smartphone and the remote display is divided into three layers:

1. Object communication layer.
   a. Purpose:
      i. Allow Java objects from the Smart Phone application to transparently map into structures on the remote device.
   b. Implementation:
      i. Smart Phone:
         1. Each object has a function which serializes the object.
         2. When properly packed by the above function, the object can be sent as a series of bytes over the Data transfer layer
      ii. Remote Display Device:
         1. Data from the Data transfer layer is in memory, organized as a list of C structures.
         2. Each C structure contains the data from each object on the smart phone.

2. Data transfer layer.
   a. Purpose:
      i. Allow sequences of bytes from the Smart Phone to be written into the buffer on the remote display device.
   b. Implementation:
      i. Smart Phone:
         1. Add header to objects serialized by the Object communication layer
            a. Header includes number of bytes to be transferred as well as address in memory where those bytes are to be written.
      ii. Remote Display Device:
1. Header is received first from the Bluetooth stack layer. Receiving the header configures the remote display device to write N subsequent bytes into the address specified in the header.

   a. Purpose:
      i. Allow sequences of bytes to be reliably transferred between Android and Remote Display device.
   b. Implementation:
      i. Smart Phone:
         1. The Bluetooth stack is part of the Android operating system.
      ii. Remote Display Device:
         1. The Bluetooth stack is part of the RN42 chip used for Bluetooth communication. The MCU uses UART to communicate with the RN42 chip.

**Firmware Implementation:**
The Firmware running on the Remote Display device has the following main roles:

<table>
<thead>
<tr>
<th>Role</th>
<th>Trigger</th>
<th>Required Response Time</th>
<th>Task execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Bluetooth data</td>
<td>Bluetooth data arrives</td>
<td>&lt;10 micro seconds. The microcontroller communicated with the Bluetooth module using UART. The MCU stores 1 byte of UART data inside a register, and that data must be read before the next byte arrives. Max supported UART speed is 928K, so up to 115,200 bytes arrive every second, requiring a response time of less than 10 micro seconds.</td>
<td>&lt;10 micro second In order to guarantee a 10 micro second response time, the task must always finish in less than 10 micro seconds.</td>
</tr>
<tr>
<td>Draw data from buffer</td>
<td>Touch input, new data to draw</td>
<td>&lt;100ms In order to provide a smooth end user experience, the</td>
<td>&lt;100ms In order to be able to be able to guarantee a 100ms</td>
</tr>
</tbody>
</table>
The drawing of data from the buffer must be quick.

redraw response time, the task must finish in less than 100ms.

| Respond to touch input | Display is touched. | <100ms The limit for touch response time is set at 100ms to minimize perceived lag to user touch input. | <100ms The touch response task must be completed quickly to allow the next touch response task to execute. |

Table 2: Roles of the remote display firmware.

In order to guarantee the timing for the three roles, the firmware had to be structured appropriately. The code architectures considered were: Round Robin with Interrupts and a Real Time Operating System. A simple polling loop was not considered as a viable option; because it would be very difficult meet the timing requirements with such an implementation.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Round Robin With interrupts</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pros/Cons</td>
<td>Can meet timing requirements. May be difficult to add another real-time task if the need arises. Gives flexibility to do whatever is needed to make the firmware work. Straightforward implementation without the need for additional software.</td>
<td>Can meet timing requirements. Easy to add additional real-time tasks if the need arises. Locks the implementation into a specific structure. Significant time must be invested to locate a suitable RTOS and learn how to use it.</td>
</tr>
</tbody>
</table>

Table 3: Pros/Cons of possible system architectures

Both the Round Robin with interrupts and the RTOS look to be suitable for the Remote Display device, but according to “An Embedded Software primer”, one should “Select the simplest architecture that will meet your response requirements.” (Simon 132), so the Round Robin with interrupts was chosen as the system architecture.

Sequence Diagram showing high level firmware operation.

As stated in table XX, the firmware has three main tasks to accomplish. This section will discuss in detail how each of the tasks was implemented, and finally how each task interacts with the others.
Bluetooth Communication Implementation

As discussed in the Hardware prototype implementation section, the RN42 Bluetooth module used in this project handles all of the low level Bluetooth communication and is connected to the STM32 MCU via UART. Bytes sent from the smartphone via Bluetooth arrive serially at the MCU, and they must be placed in memory. The STM32 has been configured to trigger the USAR1_IRQHandler interrupt when a byte is received. The following state machine has been adopted to handle the data:

Figure 7: UART input state machine for Bluetooth communication

As illustrated in the state machine, the first six bytes of data from the smartphone are a header which contains the address and the number of bytes to be transferred. After the address and number of bytes have been received in the “Get Addr” and “Get nBytes” states, the system moves to the “Get Data” state where the specified amount of data gets received and written to memory. After all of the bytes are transferred, the state machine returns to its original state waiting for an interrupt to occur.

Due to the strict timing requirements of the UART transfer as well as the simplicity of the operation, the entire state machine has been implemented in the interrupt service routine of the USART1_IRQHandler.

DMA was considered as an alternative to the above state machine. DMA would have allowed the required data to be placed in memory without the need for the MCU to be constantly interrupted. In most cases, DMA is the simplest and most efficient method of getting data from peripherals, but in this case, the use of DMA would have been problematic. The object communication protocol is implemented in a way which requires data to be written to specific addresses in the MCU memory. The specific address, and number of data bytes is sent at the same time as the data, so the window of time to configure the DMA is quite short, possibly violating the UART timing requirement. This alternative may have been viable, but the number of bytes being transferred from the
smartphone to the remote display device is typically not very large, so configuring DMA to perform each transfer would likely have been more costly than to simply use interrupts.

Another alternative considered was to use DMA to copy all UART data into a buffer, and once the entire transfer completed, process the data and distribute it to the appropriate location. The main disadvantage of this approach is that extra memory would be required for the buffer and the number of bytes that could be transferred in a single transaction would be limited by the size of that buffer.

**DISPLAY TASK IMPLEMENTATION**

The purpose of the Display Task is to draw Remote Display Objects onto the LCD screen. From a high level point of view, the display task iterates through the list of Remote Display Objects contained within the Remote Display Object Buffer, and for each object determines how and where the object should be drawn, taking the various object parameters into account.

This section of the paper will discuss the implementation of the display task from the high level down to the details of drawing individual pixels.

In the design phase of the project, it was decided that a simple form of vector graphics would be used for drawing Remote Display objects and that all scrolling and zooming of the objects was to be implemented on the Remote Display Device.

The scrolling and zooming features were implemented by using the concept of a world and of a view. In the implementation, the world has a 16bit coordinate system allowing for $2^{16} \times 2^{16}$ pixels of data. However, the display can only display 240x320 pixels at a time, so only a small portion of the view is visible at any given time. When a Remote Display Objects needs to be scrolled, the view of the world changes, moving the location of the Remote Display Object shown on the display.
Figure 8: Demonstration of basic scrolling of a text message.

Zooming of the Remote Display objects is accomplished in a similar manner. The world coordinates are scaled as needed to either show more or less of the remote display object in a given view.
A change in the view caused by user interaction with the touchscreen, or a redraw command from the smartphone.

The actual process of drawing the remote display objects is accomplished by examining the parameters contained within each Remote Display object and following the commands contained in the “Draw Command Buffer” of the object. The Draw Command Buffer simply contains a list of points in the world coordinate system which are to be connected with lines of the specified color.

Using a graphics library for handling the low level drawing was strongly considered, but the decision was made to implement an extremely simple drawing function instead. In the early stages of the project, the decision was made that the only absolutely necessary drawing function was the function to draw a line between two points on the display. Many excellent graphics libraries exist for the Cortex-M3 architecture, and they offer impressive looking widgets, radio buttons, text boxes and other relatively high level drawing functions, but integrating one of the libraries just to have access to a function to draw a line would not have been worthwhile. Instead, a line drawing function based on Bresenham’s algorithm was used. The implementation was closely based on the Pascal example code from (Kennedy). Bresenham’s algorithm was chosen because it can compute the location of pixels required to form a line between two points efficiently and without the need for complex floating point arithmetic.
The display on the Cute Digi board uses an ILITEK ILI9325 LCD Single Chip Driver for controlling the TFT display. The ILI9325 contains a buffer for all of the pixel values and is connected to the STM32 MCU via a 16-bit RGB interface. In order to change the value of a specific pixel, the address of that pixel must be set, and the RGB value of that pixel written to the buffer. Because a display buffer exists on the ILI9325 chip, the decision was made to not buffer any display data on the STM32 chip. Instead, all of the pixel data is recalculated each time a redraw operation is requested.

**TOUCH INPUT IMPLEMENTATION**

The Touch Screen on the Cute Digi board is a 4-wire resistive touchscreen connected to a TSC2046 Touch Screen Controller. The touch screen controller, in turn is connected to the STM32 chip via SPI. In addition to the SPI interface, an interrupt request line is connected between the touch screen controller and the STM32. This interrupt line goes low when the touch screen controller detects that the screen has been pressed. In the remote display device, this interrupt line is used to eliminate polling the touch screen to check if it has been pressed.

Once the touchscreen interrupt has been triggered, the following state machine governs the touch screen operation:

![State chart showing the touchscreen input state machine.](image)

Figure 10: State chart showing the touchscreen input state machine.
The above state machine is implemented in the interrupt service routine for the touch screen interrupt. An external interrupt line of the CPU is connected to the touchscreen controller, and when the screen is pressed the line goes low triggering an interrupt. The ISR for the touch screen starts a 50ms timer which is used to poll the touch screen position at regular intervals and advance the state machine as needed. After the state machine reaches its final state the timer is disabled, and the state machine starts over when the screen is pressed.
**ANDROID API IMPLEMENTATION:**
The Android side of the Remote Display API is implemented as an Android Library to allow any application to link to and use the API.

The following class diagram represents the API implementation:

![Class Diagram](image)

Figure 11: Class Diagram showing basic API structure
The following sequence diagram demonstrates the initialization of the API as well as the display of a single object on the remote display:

Figure 12: RemoteDisplayAPI initialization and the addition of a RemoteDisplayObject
The following sequence diagram demonstrates how a callback is processed by the Bluetooth read thread.

Figure 13: Sequence diagram showing a callback from the remote display device.
APPLICATION IMPLEMENTATION/API TESTING:
The implementation of the Bike Map application was done in three parts: Bike
Map/Speed display, Text Message Display and Text Input mode.

Upon creation of the Android activity a RemoteDisplayAPI object is created. This
establishes a Bluetooth connection to the Remote Display Device, and initializes the
internal structures. Next, various RemoteDisplayObjects were added to the
RemoteDisplay device, and their appropriate properties were set.

The components of each screen were laid out on the remote display device using the
SetXPos and SetYPos functions in addition to the SetHeight and SetWidth functions. The
commands to draw each object were added using the SetDrawCommands function of the
Remote Display API.

Bike Map/Speed display
The implementation of this part of the application started with accessing a .GPX file from
the internal memory of the cell phone. The .GPX files were acquired from
VenturaCountyTrails.org (VenturaCountyTrails.org). The .GPX file was formatted as
follows:

<trk>
<name>Los Robles West Singletrack</name>
<src>DeLorme Topo USAÂ® 8.0</src>
<trkseg>
<trkpt lat="34.170291424" lon="-118.888893127">
<ele>258.850769</ele>
<time>2009-07-10T22:19:39Z</time>
<fix>3d</fix>
</trkpt>
<trkpt lat="34.170248508" lon="-118.889021873">
<ele>259.914520</ele>
<time>2009-07-10T22:19:39Z</time>
<fix>3d</fix>
</trkpt>

Each track represents a trail, and each trkpt represents a point on the trail. A trail is drawn
by connecting a series of trkpt’s on the remote display. A number of such GPX files used
together form a map of a trail system. In order to fit the data onto the small Remote
Display device, each trkpt latitude and longitude is converted into a point in a
drawCommands buffer and is scaled into the world 2^16X2^16 coordinate system. The
scaling is accomplished in a manner which sets the south most and west most point of the
map in the bottom left corner of the remote display device. Once the list of points has
been properly scaled, each set of drawCommands is added into the drawCommand buffer
of a remote display object, the objects scrollable and visible properties are set and the
objects all get drawn onto the Remote Display Device.
With the drawing of the base map accomplished a thread is started which asks the Android system for location data. When a new set of coordinates arrives from the operating system, the coordinates are scaled to the world coordinate system, and the red marker representing the users’ location is drawn onto the display. The coordinate and time data are also used to calculate the users speed and to display it in the upper right hand corner of the display device.

Picture of map with marker and speed

Figure 14: Map display mode of application. Showing West Los Robles Trail in Thousand Oaks and a speed of 13MPH
**Text Message Display**

When the user presses the text message icon in the upper left hand corner of the display, the application switches to Text Message Display Mode. In this mode, instead of displaying the map and speed a text message is shown on the display. Each letter in the text message is simply a set of drawCommands in the drawCommands buffer of the text display object. The drawCommands for the text are generated using a function which takes as input a string of characters and outputs a set of properly formatted drawCommands. In addition to the text message, two buttons are displayed in this mode. The Close button closes the Text Message Display mode and the Reply button opens the Text input mode of the application. Each button is implemented as a separate remote display object with its own callback method.

![Sample text message displayed on remote display device.](image15)

![Zoomed in text message display.](image16)

Figure 15: Sample text message displayed on remote display device.

Figure 16: Zoomed in text message display.
**Text input mode**

This mode of the application is intended to allow the user to respond to text messages. After the user has clicked the Reply button in the Text Message display mode, the on screen keyboard is displayed. The on screen keyboard is a simplified version of the 8pen (8pen) smartphone keyboard, and it consists of five regions. Each region is represented as a separate RemoteDisplayObject and each region has a callback method which is triggered when that region is clicked. When the center region is clicked, the sequence of object clicks is checked against a list of valid sequences, and if a match is found, the correct character is added to a string, which is displayed in the upper left quadrant of the display.

In addition to the keyboard, the text input mode of the application has two buttons: Back and Reply. The keyboard can be difficult to use, so mistakes are often made, and the Back button allows the user to erase the last typed character. Upon completion of the text entry, the user presses the Reply button to reply to the original text message which was displayed in the Text Message Display mode of the application.

![Figure 17: Text input mode of the remote display device](image)

Figure 17: Text input mode of the remote display device
APPLICATION PERFORMANCE ANALYSIS:

Application/Firmware Timing
The following set of figures were captured using the IAR Systems Embedded Workbench for ARM and the i-Jet debug probe. The interrupt timing data and data values shown in the graphs is derived from the Serial Write Output (SWO) of the Cortex-M3 processor.

Figure 18: This figure shows the draw time for the MAP in the sample application. Current_object_index represents the index of the object currently being drawn. The elapsed time from the start of the drawing operation to the end of the drawing operation is just over 10ms.

Figure 19: This figure shows the input of the letter “Y” in the text input mode of the application. The last_click variable shown above represents the callback_number of the last object to be clicked. At 399.6s, the callback number changes from 10 to 11, indicating a click in region one of the onscreen keyboard. Next, region 4, 3 and 0 are all clicked, indicating the letter “Y”. At ~400.08s the final byte of data indicating that region 0 was clicked gets sent and at ~400.16s the operation to write that letter to the screen is
completed, resulting in an elapsed time of 18.6ms for the transfer and drawing of a character in the text input mode.

Figure 20: This figure shows the time required to perform a close text message operation. At 310.62s, the user first pressed the touchscreen in the location of the close button, and at 311.00s the drawing of the updated display data was completed, resulting in an elapsed time of 380ms.

Figure 21: This figure shows the nesting of interrupts on the Remote Display Device. TIM3 represents the touchscreen polling timer which triggers an interrupt every 50ms after the screen has been pressed, and USART1 represents the Bluetooth communication interrupt which occurs every 10 micro seconds. As can be seen above, many USART1 interrupts occur before the TIM3 interrupt service routine finishes.
Figure 22: This figure shows that the USART1 interrupt service routine meets the timing requirements. The service routine completes in less than 1.667 micro seconds, leaving plenty of time before the next UART byte arrives in 10 micro seconds.

Power consumption of remote display device

As can be seen in figures 18-22, the Remote Display Device consumes ~200mA of current during normal operation. During the above measurements the device was powered by the 4.7V power output of the I-Jet debug probe, resulting in a power consumption of ~0.94 watts.

Unfortunately, such power consumption is unacceptable for the Remote Display Device. Typical small cell phone batteries provide ~4 watt hours, giving the Remote Display Device a short 4 hour battery life.

There are several reasons for this short expected battery life:

1. LCD display uses a powerful LED backlight.
   a. A backlit LCD display is not the appropriate display for a device which is expected to operate for extended periods of time. A transflective display, which does not requires a backlight to operate would have been a much better fit for this project, but such displays are not as readily available with a touchscreen as normal backlit LCDs.

2. Inefficient board power circuit
   a. The Mini STM32 board was designed to be powered by a 5V USB port, not by a battery. The 5V input gets brought down to 3V using a linear regulator, wasting as much as 40% of the energy, and heating up the board significantly.
3. Bluetooth communication is not optimized.
   a. No time was put into optimizing the Bluetooth communication between the
      Smartphone and the Remote Display Device. Various Bluetooth power saving
      modes, such as sniff mode, lowering the transmission power, and putting the
      device to sleep when not needed exist, but they are not used in this project due
      to development time constraints.
TOOLS USED

Embedded Workbench
The IAR System Embedded Workbench for ARM was used for the development of firmware for the Remote Display Device. The Embedded Workbench is an IDE which includes a compiler, debugger as well as example projects for the STM32 CPU. The Embedded Workbench made it easy to compile and load the firmware onto the CPU.

During development and debugging of the firmware, the SWO display feature of the Embedded Workbench and the I-Jet probe proved to be invaluable. It was possible to see all of the interrupts occurring on the CPU along with the values of select variables. Using this feature it was easy to identify that the interrupts are correctly nested and to measure the performance of various parts of the application.

![Embedded Workbench for ARM](image)

Figure 23: Embedded Workbench for ARM shown during debugging of the firmware

Eclipse
For the development of the Android side of the project, the Eclipse IDE with the ADT (Android Development Tools) plug-in was used. The plugin made it possible to create Android Activates and to run them in both the phone simulator and on a real smartphone.
Figure 24: Showing Eclipse IDE during development of the demo application.

**Phone Simulator**

In the early stages of development, the Android application was ran on a simulated phone image created by the ADT plugin for eclipse. Unfortunately, the Phone Simulator does not have support for Bluetooth connections, so it was necessary to have the phone simulator connect to the PC via a loopback TCP/IP interface, and have the PC act as a proxy and transmit the bytes over its Bluetooth antenna. Performance of the Phone Simulator was not satisfactory. For example, the loading of several KB of map data from XML files took as long as 1 minute. This slow performance forced the switch to using a real smartphone as the development platform for the Android API and application.

**Samsung Galaxy S**

A Samsung Galaxy S smartphone was used in the later stages of the API and application development. The real smartphone provided much better performance than the simulated version, performing tasks up to 20X faster. Thanks to the seamless integration with the Eclipse IDE it was easy to install and execute the application on the smartphone.
CONCLUSION
In conclusion, this project was successful at meeting its goal of providing a working prototype Remote Display for SmartPhone device, but it came well short of offering a final implementation of a marketable product. The project proved to be a great learning experience and allowed me to strengthen my knowledge of embedded systems programming and the development of Android applications.

Lessons Learned
One of the most difficult parts of this project was the fact that three separate parts of the project were being developed concurrently by one developer. As a new feature was added to the firmware of the Remote Display Device, the feature was added to the Remote Display Device API on Android, and also added to the Demo Application. This resulted in frequent switching of development tasks, programming environments and languages and led to three different parts of the system being debugged at the same time. Instead, it would have been more effective to develop and test each part of the system independently with well-established interfaces and combine the different parts once they were proven to function as expected. For example, instead of relying on the Bluetooth data from the smartphone to get to the remote display device to verify that the drawing commands for the letter ‘A’ are correct, it would have made more sense to write a simple Android application which behaved how the remote display device should.

Further Work
A number of areas of this project can be developed further:

1. Add additional vector draw commands
   a. All shapes can technically be drawn using lines, but the drawing of a filled in rectangle, or a circle would require too many lines to be feasible.
   b. The additional draw commands would enable much prettier graphics on the remote display device.
   c. This could be accomplished by using an existing Cortex-M3 graphics library.
2. Use an LCD display more suited for the task.
   a. The LCD display from the Mini STM32 board draws too much current to be a good candidate for a Remote Display Device such as this.
3. Optimize power consumption
   a. Currently, the power consumption of the prototype device is too high to be usable as intended. Battery life would likely be limited to ~4h on a large Li-ion battery.
      i. Use efficient display type.
      ii. lower Bluetooth transmission power and use sniff mode.
      iii. Optimize firmware to use lower CPU frequency and sleep modes.
BIBLIOGRAPHY


Kennedy, John. *Bresenham's Integer Only Line Drawing Algorithm*. Santa Monica, CA: Mathematics Department, Santa Monica College, n.d.


VenturaCountyTrails.org. 10 June 2011. April 2013.