Low-Power Self-Sustaining Schedule Display

A Senior Project

presented to

the Faculty of the Electrical Engineering Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Electrical Engineering

by

Sam August

June, 2011

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Thanks to everyone who supported me, especially Mom and Dad.

I. INTRODUCTION

The completed project is an Amazon Kindle, powered by a USB solar charger, running a student designed program to display a weekly schedule for a faculty member. The purpose of the project was to design a sustainable electronic schedule display for use outside a faculty office. It was designed in the hopes that faculty members could more easily update their schedules, and possibly allow for students to schedule appointments. The program was designed using Java and the solar-charger was designed using a solar cell, Schottky diode, and a USB to micro-B USB cable.

II. BACKGROUND

The idea for the project originally resulted from the desire for an electronic low-power PDF reader. As we began to consider it more and more as a possibility, this desire manifested into the idea for a low-power, sustainable schedule display. Ideally the final product would mount outside a faculty office and replace the current system of a printed piece of paper. The hope would be that this would be easier for a faculty member to update their changing schedule from week to week, as well as remain open for future development of functionality such as a touch-screen or student access to make appointments.

III. REQUIREMENTS

In order for the project to have some level of coherence, a listing of guidelines for the design was created. First of all we wanted to make the project sustainable. Since the item seemed like more of a luxury than a necessity, we did not feel justified in designing a power-hungry system. As well, self-sustained systems are more cost-effective in terms of long-time use, and thus seemed more appealing. The first major requirement was to make sure that the design could electronically display an entire week's schedule. The project also needed to be relatively inexpensive. Since it was not sponsored by any outside sources, any investment came purely from our own wallets. As a result, the project needed to stay within our own budgets. It was important for the design to have a level of interactivity as well. Incorporating this was necessary to make the device easier to use for both students and faculty. As well, by making the project interactive it stands out much more than the previous technology of stapling a paper to the bulletin board. An interactive device also begins the steps for future development to expand and implement functionality such as student appointments input. Because the final goal for the project is to mount outside a faculty office, the design also needed to be light enough so it can successfully hang on the wall. Finally, the design also required easy updating of the schedule. If each time the schedule needed to change programs had to be re-compiled and reloaded onto the device, it would not be very useful for the owner. These requirements compiled together helped to lay the foundation for the design process.

IV. DESIGN

The design process took many steps that would build on each other. With a foundation of requirements, the next step was to decide what physical components were needed for the final product. To view the schedule an electronic display was necessary, and including interactivity required some type of buttons or switches. As well, the idea for keeping the system sustainable was inclined to manifest in some sort of solar charging, so we knew a solar cell would be included in some fashion. At this point, it seemed the next step would be to decide on a display.

A. Choosing the Display

Finding a suitable display was an important step in the development of the project because it would determine how to confront the remainder of the work. Many display devices were possible, although the choice between two types seemed important to focus the decision: a regular LCD screen, or the new e-ink technology found in the Amazon Kindle and several other products. Keeping in mind the goal to develop a sustainable project, the design leaned more in the direction of an e-ink type display. Since the project would most likely remain unused for much of the day, it seemed that an LCD screen would be a little unnecessary. As well, during use the e-ink screen uses much less power, since the only time the battery is used occurs when something on the screen is reprinted. Among e-ink screens, however, several choices were available. In order to easily evaluate all of these choices, a listing of each device with pros and cons was developed, as well as a matrix comparing device

options to proposed requirements. Both can be found in can be found in figures 1 and 2, respectively.

Option #1: Esquire Magazine 75th Anniversary E-Ink Cover

Pros:
- Inexpensive
- Could allow us more time to research a final display solution without significantly delaying progress
- Others have programmed these, so we know it can be done

Cons:
- Hardware design may not resemble that of our final choice

Option #2: 6" Touch Screen Digital Ebook Reader

Pros:
- Touch screen capability
- Decent sized display

Cons:
- No knowledge of manufacturer
- Almost 1 month to ship
- Risks involved in extracting e-ink display from device

Option #3: Amazon Kindle

Pros:
- Easy to acquire
- Popularity may make it easier to find info on the display

Cons:
- Extracting the display could still be a problem

Option #4: E-Ink AM350 Prototyping Kit

Pros:
- Exactly what we're looking for

Cons:
- Far exceeds budget limitations ($3000)

Figure 1: Comparison of e-ink display possibilities

	Esquire Mag. E-Ink Cover	6” Digital Ebook Reader	Amazon Kindle	E-Ink AM350 Prototype Board
Sustainable (2)	5	5	5	5
Interactive capability (1)	3	5	5	3
Easy to update schedule (2)	2	2	4	2
Mount outside an office (2)	5	5	5	3
Easy to program (1)	5	2	5	5
Reasonable price (2)	4	3	3	1
Weighted Total	40	37	**44**	30

Figure 2: Matrix comparing display choices to proposed requirements. Requirements are listed on the left side with weightings for each.

After some evaluation of the list, the Amazon Kindle was selected for the project’s display device. Some research uncovered that the Kindle had a well developed hacker community which boasted its capabilities as a Linux device. This was extremely attractive for the project’s design, since it meant that not only would the projected disadvantage of extracting the display from the Kindle no longer be an issue, but the inclusion of button functionality would solve another step in the initial design plan.

B. Solar Charging

To keep the project sustainable, the design would include a solar-charging device. Interfacing this with the Kindle proved to be a rather simple task, since the

Kindle has a free micro-B USB port. Research showed that the Kindle has a 3.7V, 1750 mAh battery, which Amazon claims can sustain on a full charge for a full month as long as the wireless capability of the device is switched off. From these specifications a suitable solar cell could be purchased to provide power to the battery. Eventually the decision came to a 200mA rated solar cell from an oversea supplier. This model was selected because it would provide enough current to provide a full charge in approximately nine hours (assuming perfect conditions), and would be small enough to not be obtrusive to the user's experience. Because the Kindle provides access to its battery through the USB port, the design of the solar cell was very simple. It consisted purely of the selected solar cell, a Schottky diode, a USB to micro-B USB cable, and a small project box to house the components. A diode was included in the design to prevent any possibility of current flowing from the battery to the solar cell. Specifically, a Schottky diode was selected because it would provide a lower voltage drop between the solar cell and the Kindle battery, so less sunlight would be necessary to activate the charging process. A 15-foot USB cable was used so that the final device could be mounted outside an office and the solar charger could be placed near a window. A full listing of the selected parts and their prices can be found in Appendix A. The regular USB portion of the cable would be removed to expose the power and data lines. The diode would be soldered to the positive power line with the correct current direction in mind, and the solar cell's positive output would solder to the other end of the diode. The two negative terminals of the solar cell and USB cable would be soldered together to complete the circuit. Overall, the

design was very simple, but necessary in order to meet the goal of keeping the project sustainable. Figure 3 below shows a system hardware diagram of the solar charger implementation with the Kindle.

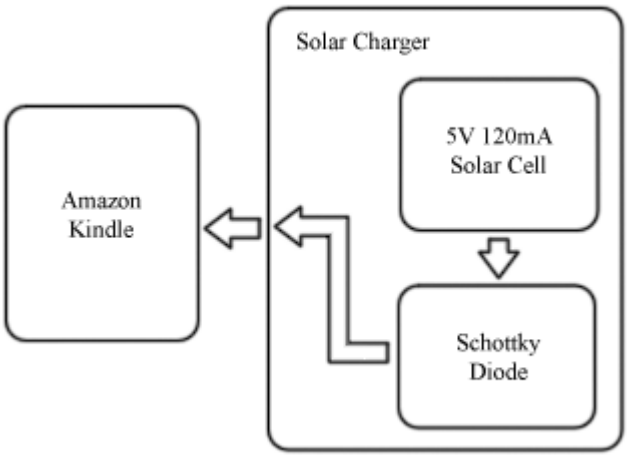


Figure 3: System hardware diagram

C. Kindle Program

Designing the program for the Kindle evolved from an attempt that, in retrospect, was not providing quality results. After acquiring the Kindle the next step was to research how to actually create content for the device. The first inclination was to attempt to import a distribution of Linux onto the device and design programs that could run through that operating system. To do this, it was necessary to use information from the MobileRead forum community, which contained a special section on hacking into a Kindle. Once the instructions provided were completed and root access to the Kindle was available, it became clear, however, that the process was not as straightforward as it had appeared. Poking around the Kindle's internals

revealed several script commands that were experimented with in an attempt to fashion the beginnings of the desired program. The problem became clear, however, that these would be extremely difficult to implement on a higher level, and their functionality was relatively limited for how much effort went into writing them. However, further research within the MobileRead forums provided information on designing programs for the Kindle using Java. After the test program was attempted, it was obvious that this was the path we would want to take to have the most functional result.

The design of the Java program was in part defined by trail and error, but once it became clear how the framework of Java programs was implemented on the Kindle, a more planned design arose. Using a GridBagLayout from the Java Standard Library, the application would contain labels across the top of the display relating to each day of the week, excluding Saturday and Sunday, of course. Underneath this the schedule information for each hour of the day would be placed. When a date label was selected, it would transfer that information into the space below the date labels, meaning only one day would be displayed at a time. This would incorporate the interactivity that the project was hoped to have, since users would need to press the buttons on the Kindle to change the days of the week they are viewing. In order to incorporate a system like this, some custom Java classes needed to be developed. One class would represent the labels for the days of the week, another would represent a schedule object, which would contain another class relating to each item, or hour of the day, in the schedule. A general schedule would be placed underneath

the day labels, and each label would reference that general schedule to populate with information for their specific day. Each label would also be assigned a specific schedule, which would contain all the information for the particular day of the week. On selection the day label would populate the general schedule with its specific schedule. The Java code for the entire program can be found in Appendix C.

The program design also needed to account for the easy updating that was desired in the requirements. Originally the idea had been to allow the user to update each piece of the schedule by making the selections editable through the Kindle's keyboard after a specific button was pressed. This, however, seemed to require more work from the user than necessary, and a simpler solution was found. Instead, the program was designed to take data from an external text file and populate the specific schedules with that information. Users could access and edit the text file by simply mounting the Kindle's user filesystem to their personal computer, and navigating to a predetermined location. Completion of the program design was the final step in completing the design of the entire project.

V. TEST PLANS

Testing of the device would be split, as expected, between the major sections of the project, namely the Java application loaded onto the Kindle, and the solar charger designed to power it. Tests consisted purely of running the application to make sure it operates correctly, and inputting various commands into the Kindle to see how they affect the program. While not the most elegant solution, it seemed the only choice for the software design. Testing of the solar cell was planned to include current and voltage ratings at different levels of luminescence. Because the unit will remain stationary, stress tests seemed to be unnecessary. As a result, much like the Java program, the tests of the solar cell were limited.

VI. DEVELOPMENT AND CONSTRUCTION

A. Kindle Program

The development of the schedule application was by far what constituted the majority of the development time in the project. It began once the Kindle was obtained from Amazon, and lasted almost until the final weeks of the project. The early development consisted mainly of finding how to gain entry to the Kindle's internal file system. This step was, in large part, a research effort through the Kindle hacker community, found on the MobileRead online forums. Some of them had developed a jailbreak for the Kindle, which allowed custom hacks to be installed on the device. As well, others had developed a hack to restore the USB networking functionality of the Kindle, which had originally been included for the use of the developers only. Re-enabling this feature would make it possible to view the Kindle as an internet device when connected to a home computer via the USB port, and from there allow for a Telnet connection which would give access the Kindle internals. A full description of the jailbreak and USB networking hacks and the procedure followed to implement them can be found in Appendix D.

Once these modifications had been made, the Kindle's internals could be scoured for any information that could give some direction towards a way to develop software. This proved to be unsuccessful, and is further detailed in the Test Results and Implementation section. Success came, however, in the form of yet another forum post. After more research on the MobileRead forums section for Kindle development, a detailed a method for designing programs on the Kindle using Java

was discovered. This was a welcome opportunity, since designing using a high-level language would allow for far greater application capabilities with much less development time.

The process of developing a Kindle program using Java first involved extracting the Java Libraries used for development. Luckily, these were stored on the Kindle, so they were easily moved to the Kindle's public folder and copied to a personal computer. Following this, a new custom keystore needed to be created for our application. A keystore is simply a storage file of security certificates that allow or disallow programs from running on the device. Understandably, the Kindle developers created their own keystore so that outside applications could not run on the device. However, with root access a custom keystore could be placed in the required directory and student designed applications could run freely. Once this was complete, development became a process of creating a Java program and testing it on the device. A more complete explanation of the process of extracting the Java libraries and creating custom keystores can be found in Appendix D.

The development of the application using Java began with simple tests. The first was to output the classic "Hello World" message to the screen, to ensure an understanding of a program's structure for use on the Kindle. Next came tests involving buttons, so two text fields were created to see if the information in them could change on a specific button press. Once again, it was a simple success. With these two baseline discoveries development could begin on what would eventually be the final application. A GridBagLayout from the Java Standard Library was used to

allow for multiple rows and columns of different sizes, and was a relatively simple layout to implement. In order to implement the day layouts as was planned in the design section, each day object needed to respond to button inputs. The problem, however, resulted in the fact that the original method to implement functionality on a button press would not work with this design. Since the custom class that was created for the labels extended a class found in the Kindle libraries (KTextArea), it had implicit in it many useable methods. Two such methods were the processKeyEvent and processFocusEvent methods. Using these, certain actions could occur on specific button presses, or in the case of processFocusEvent, in the event an object was selected with the cursor. This allowed the program to use the left and right page-turn buttons of the Kindle to scroll through day labels, and change schedules when days were selected. With the labels in place, a system for displaying schedules needed to be created. Day labels had already been given functionality for assigning a specific schedule, meaning a schedule that would hold information for that specific day, and a general schedule, meaning the schedule that would be displayed. A simple method which replaced information from the specific schedule to the general schedule was created to fulfill this. The specific schedules, however, needed to be populated with data, and that data needed to be easily changed. This is where the use of an input stream came into the design. With the Java tools already found in the Standard Library, data could be read from a given text file then parsed and loaded into the specific schedules for each date label. This method was developed in the main program code so that it would have access to all the specific schedules. The text file

is formatted such that each day of the week is listed in order, Monday through Friday, so the application can move through date by date and assign the correct data to the correct schedules. A flow chart of the program operation can be seen below in Figure 2.

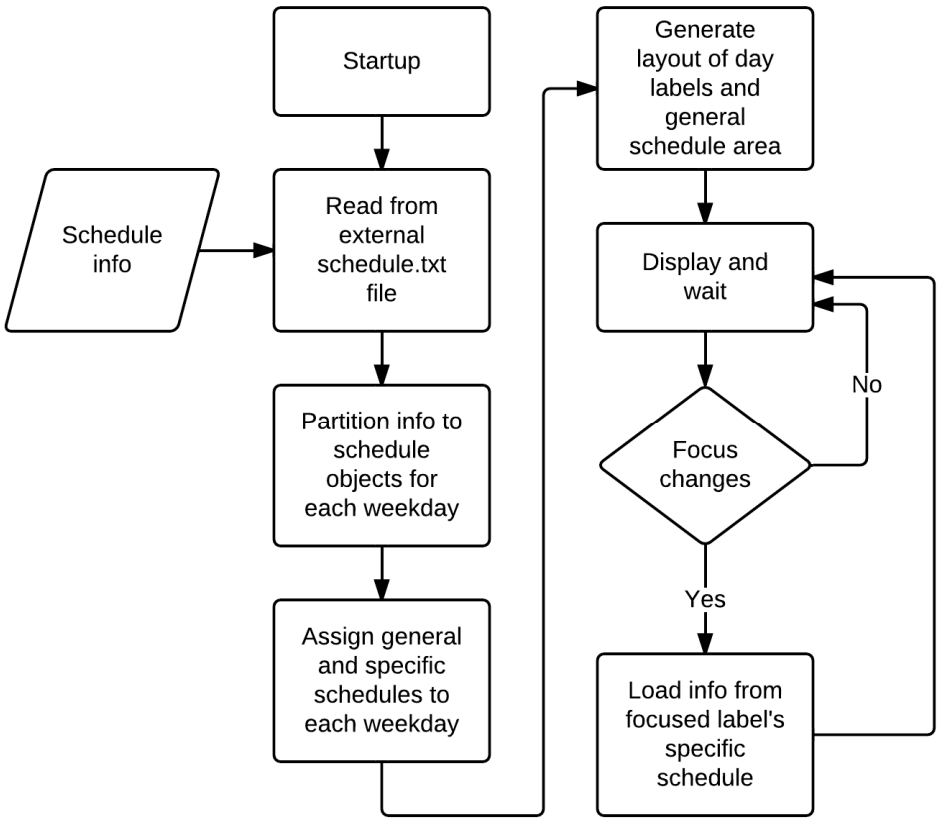


Figure 4: Flow diagram of program operation

B. Solar Cell

Development of the solar cell was rather simple, as is expected with a simple design. First the project box was altered to make space for the connections between the solar cell and the USB cable, as well as open a port on the side for the cable to escape from. These were created by simply drilling three holes in the project box, two large enough to leave the solar cell connections visible, and one small enough to snugly surround the USB cable. With these complete, the connections were soldered as described in the design and the component was tested for functionality.

VII. TESTING AND INTEGRATION

Although the testing of the two major portions of the project was somewhat limited, this did not eliminate the possibility of error and frustration. Most notably this occurs in the Java program, which could only be tested by trial and error. Many times the program was compiled only to return errors or inconsistencies with the planned design. The most troublesome error that occurred during the development resulted from attempting to read an external file to obtain schedule information. This task, though seemingly basic, was made impossible by one small requirement that was realized only after many hours of trial and error. The original design was to use the Java Standard Library classes InputStream, InputStreamReader, and BufferedReader to read and parse the information in the text file. This seemed simple enough, since previous experience with these classes provided baseline code to draw from. The problem arose, however, on running the program in the Kindle, which would result in an error message stating "An error occurred opening the title." This was particularly difficult to manage, because there was no indication on runtime as to the nature of the error. Because of this, the debugging process was tedious. At first attempts were made to read the file using different methods than InputStreamReader, including Scanner and FileReader. Both of these methods, however, proved no better at solving the issue. The search for the solution moved to the Kindle Developer Forums, which were opened when Amazon began a project to allow developers to work under a non-disclosure agreement on developing applications for the Kindle. After poking through the site, it was clear that the code we had written to get data

from the text file was, in fact, correct. This only served to cause more confusion, so Prof. Kurt Mammen, a Cal Poly Java instructor, was contacted. A meeting with him reinforced the correctness of the code. He suggested, however, taking the code line by line to find the offending piece. Thanks to him it became clear the error was occurring not when the file was acquired by the InputStream class, but instead when a read attempt occurred by InputStreamReader. With a solution lacking and a deadline approaching, the MobileRead Forums were once again used for help. After posting the problem, another user explained that the Kindle is designed to prevent applications from reading from any file outside their own work folder, found in the file system open to users. After changing the file path in my code and loading the file into the correct location, the program was up and running.

Another difficulty in the development occurred after gaining access to the Kindle through the restored USB networking function. The difficulty was less error based, and more a realization of a poor design. After exploring the internals of the Kindle, several shell scripts that had been left on the Kindle for debugging purposes were found. The scripts provided information on benchmarks such as button press latencies and screen refresh rates. The script files were used to extrapolate commands that could run in the terminal window through root access, and manipulate functions like button commands or image displays. Towards the end of the attempts to use this method, methods to present text on screen were even tested. Some serious problem arose in all of this, however. While it seemed impressive on the small scale, the larger picture of the project made it clear that the commands being experimented with

provided little in terms of functionality, while designing them was far from simplistic. As well, the question arose of how to implement these scripts to run easily on the Kindle. All those that were created by the Kindle developers had been accessed via root access through a terminal. This method was less than convenient, and while a script could be developed to run indefinitely, would not be ideal if, say, the Kindle needed to be restarted.

Difficulties also occurred in planning with the solar cell. As listed previously, the solar cell testing had originally consisted of current and voltage measurements at various levels of luminescence. The time spent attempting to program for the Kindle with script left the project rushed to meet the deadline, and as a result the testing of the solar cell was not complete at the time of this writing. It is possible that these tests could still occur, and if so will be included in an appendix to this report. The best way to avoid this would have been better time management, plain and simple. Towards the end of the project things began piling up that, in retrospect, could have been dealt with sooner. A perfect example of this is the solar cell testing. The parts were available at the beginning of the Spring quarter, but construction did not complete until nearly halfway through that quarter. If a little more time had been devoted to working on that earlier, the tests would have easily been completed and the total project would have looked more complete.

VIII. CONCLUSION

As a whole the project was a success. The original plan was to make a low-power, self-sustained schedule display, and the end result was just that. The amount of time spent attempting to work with solutions that proved to be inadequate was somewhat disappointing. As a result the final project, though it meets the original requirements, seems that it could have had more functionality in the final design had development more directly followed the final approach used. As mentioned previously, those mistakes also left testing on the solar cell incomplete, which would have been very valuable information for the user of the final device. Nonetheless, the project fulfills its intended functionality and therefore, was definitely worth the time.

IX. BIBLIOGRAPHY

"5V 210mA Power Supply Solar Cell Panel Charger battery | eBay." Internet: http://cgi.ebay.com/ws/eBayISAPI.dll?ViewItem&_trksid=p4340.l2557&hash=item336565bf54&item=220744499028&nma=true&pt=LH_DefaultDomain_0&rt=nc&si=p9%252BjkTUYitqQo7zzPE10j3pEX2A%253D#ht_8931wt_1002, March 25, 2011 [June 9, 2011].

"Amazon Kindle – Wikipedia, the free encyclopedia." Internet: http://en.wikipedia.org/wiki/Amazon_Kindle, May 26, 2011 [May 26, 2011].

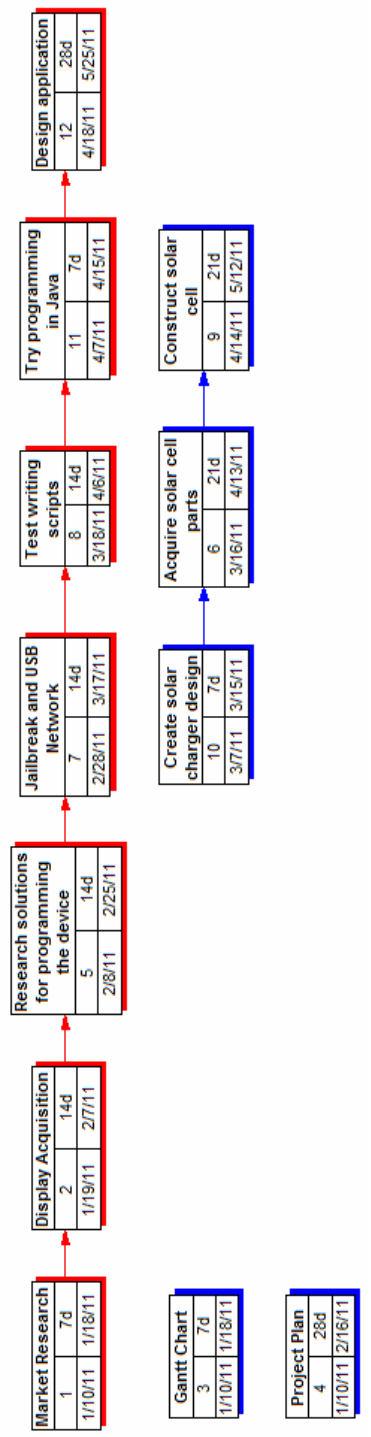
KukMan. "Guide: How to write Kindlets." Internet: http://www.mobileread.com/forums/showthread.php?t=102386, March 15, 2011 [May 27, 2011].

"Schottky diode – Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Schottky_diode , June 2, 2011 [June 9, 2011].

APPENDIX A: Parts List

Amazon Kindle 3 (Wireless Version)	$139
200 mA Solar Cell	$15
15ft USB to USB Micro-B Cable	$10
Low Voltage Schottky Diode	$1.20
Desktop Enclosure (5.51” x 4.33” x 1.38”)	$7.15

APPENDIX B: Schedule



APPENDIX C: Program Listing

Main.java

```
/**
 * Main.java
 * Developed Spring Quarter 2011
 *
 * Description: Main file for display and control of a scheduling program
 *
 * @author Sam August
 */

package schedule;

import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;

import com.amazon.kindle.kindlet.AbstractKindlet;
import com.amazon.kindle.kindlet.KindletContext;
import com.amazon.kindle.kindlet.ui.KPanel;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;

public class Main  extends AbstractKindlet {

    private KindletContext ctx;

    // Create date labels
    private dateItem monday = new dateItem("Monday", 0);
    private dateItem tuesday = new dateItem("Tuesday", 1);
    private dateItem wednesday = new dateItem("Wednesday", 2);
    private dateItem thursday = new dateItem("Thursday", 3);
    private dateItem friday = new dateItem("Friday", 4);

    //Create general and specific schedules
    private schedule schedule = new schedule();
    private schedule monSchedule = new schedule();
    private schedule tueSchedule = new schedule();
    private schedule wedSchedule = new schedule();
    private schedule thuSchedule = new schedule();
    private schedule friSchedule = new schedule();

    // Note: compiling with concatenation has caused errors previously.
    //       Presented as such here purely for readability.
    private String path = new String("/mnt/us/developer/Johns Schedule"
                                     + "/work/schedule.txt");

    // getSchedules Method:
    // Takes data from external file and populates specific schedules
    public void getSchedules(schedule monday, schedule tuesday,
                             schedule wednesday, schedule thursday,
                             schedule friday) {

        try
        {
            // When getting data from an external file, that file MUST
            // be placed in the "work" directory of the program and
            // can only be read from that location
            InputStream stream = this.getClass().getResourceAsStream(path);
            InputStreamReader reader = new InputStreamReader(stream);

            BufferedReader bufferedReader = new BufferedReader(reader);
```

```
        String word = new String();
        int i = 0;

        // Ignores the line that specifies the day in the text file
        bufferedReader.readLine();

        // Each day supports 15 hours.  Reads through each hour in the
        // external file
        for (i = 0; i < 16; i++)
        {
            word = bufferedReader.readLine();
            monday.setDate(i, word);
        }

        // Repeat the previous pattern for each day of the week
        bufferedReader.readLine();

        for (i = 0; i < 16; i++)
        {
            word = bufferedReader.readLine();
            tuesday.setDate(i, word);
        }

        bufferedReader.readLine();

        for (i = 0; i < 16; i++)
        {
            word = bufferedReader.readLine();
            wednesday.setDate(i, word);
        }

        bufferedReader.readLine();

        for (i = 0; i < 16; i++)
        {
            word = bufferedReader.readLine();
            thursday.setDate(i, word);
        }

        bufferedReader.readLine();

        for (i = 0; i < 16; i++)
        {
            word = bufferedReader.readLine();
            friday.setDate(i, word);
        }

        // Make sure to close all inputs
        bufferedReader.close();
        reader.close();
        stream.close();
    }



    catch (Exception e)
    {
        throw new RuntimeException(e);
    }

}

public void create(KindletContext context) {
        this.ctx = context;
        final KPanel mainPanel = new KPanel(new GridBagLayout());

        GridBagConstraints gc = new GridBagConstraints();
```

```
// Get schedule data from external file
this.getSchedules(monSchedule, tueSchedule, wedSchedule,
                  thuSchedule, friSchedule);

// Sets all the schedules
monday.setSchedule(schedule);
monday.setSpecific(monSchedule);
tuesday.setSchedule(schedule);
tuesday.setSpecific(tueSchedule);
wednesday.setSchedule(schedule);
wednesday.setSpecific(wedSchedule);
thursday.setSchedule(schedule);
thursday.setSpecific(thuSchedule);
friday.setSchedule(schedule);
friday.setSpecific(friSchedule);

// Set up specifications for layout
gc.gridx = 0;
gc.gridy = 0;
gc.anchor = GridBagConstraints.NORTHWEST;
gc.weightx = 0.1;
gc.weighty = 0.1;

// Draw day labels
mainPanel.add(monday, gc);

gc.gridx = GridBagConstraints.RELATIVE;
mainPanel.add(tuesday, gc);
mainPanel.add(wednesday, gc);
mainPanel.add(thursday, gc);
gc.gridwidth = GridBagConstraints.REMAINDER;
mainPanel.add(friday, gc);

// Draw schedule area
gc.gridy = 1;
mainPanel.add(schedule.sevenAM, gc);

gc.gridy = 2;
mainPanel.add(schedule.eightAM, gc);

gc.gridy = 3;
mainPanel.add(schedule.nineAM, gc);

gc.gridy = 4;
mainPanel.add(schedule.tenAM, gc);

gc.gridy = 5;
mainPanel.add(schedule.elevenAM, gc);

gc.gridy = 6;
mainPanel.add(schedule.twelvePM, gc);

gc.gridy = 7;
mainPanel.add(schedule.onePM, gc);

gc.gridy = 8;
mainPanel.add(schedule.twoPM, gc);

gc.gridy = 9;
mainPanel.add(schedule.threePM, gc);

gc.gridy = 10;
mainPanel.add(schedule.fourPM, gc);

gc.gridy = 11;
mainPanel.add(schedule.fivePM, gc);
```

```
            gc.gridy = 12;
            mainPanel.add(schedule.sixPM, gc);

            gc.gridy = 13;
            mainPanel.add(schedule.sevenPM, gc);

            gc.gridy = 14;
            mainPanel.add(schedule.eightPM, gc);

            gc.gridy = 15;
            mainPanel.add(schedule.ninePM, gc);

            gc.gridy = 16;
            mainPanel.add(schedule.tenPM, gc);


            try {
                // Create entire layout by adding mainPanel
                ctx.getRootContainer().add(mainPanel);

            } catch (Throwable t) {
                    t.printStackTrace();
            }
    }

}
```

****

dateItem.java

```
/**
 *
 * @author Sam
 */
package schedule;

import com.amazon.kindle.kindlet.event.KindleKeyCodes;
import com.amazon.kindle.kindlet.ui.KTextArea;
import java.awt.event.FocusEvent;
import java.awt.event.KeyEvent;

public class dateItem extends KTextArea {

    protected boolean toggle = true;
    protected int index = 0;
    protected schedule dateSchedule = new schedule();
    protected schedule specificSchedule = new schedule();

    public dateItem() {
        this.setEditable(false);
    }

    public dateItem(String date, int val) {

        this.setText(date);
        this.index = val;
        this.setEditable(false);
        this.setSize(120,20);
    }

    public void setIndex(int val) {

        this.index = val;
    }
```

```
    // Sets the general schedule that data will be sent to
    public void setSchedule(schedule input) {

        this.dateSchedule = input;
    }

    // Sets the specific schedule that data will be gathered from
    public void setSpecific(schedule input) {

        this.specificSchedule = input;
    }

    // Reassign key commands so page turn works as a fiveway right or left
    // command.  Alters the caret position so page turn does not move letter
    // by letter.
    protected void processKeyEvent(KeyEvent e) {

        if (e.getID() == KeyEvent.KEY_PRESSED &&
            e.getKeyCode() == KindleKeyCodes.VK_RIGHT_HAND_SIDE_TURN_PAGE)
        {
            e.setKeyCode(KindleKeyCodes.VK_FIVE_WAY_RIGHT);

            this.setCaretPosition(this.getText().length());

            super.processKeyEvent(e);
        }

        if (e.getID() == KeyEvent.KEY_PRESSED &&
            e.getKeyCode() == KindleKeyCodes.VK_TURN_PAGE_BACK)
        {
            e.setKeyCode(KindleKeyCodes.VK_FIVE_WAY_LEFT);

            this.setCaretPosition(0);

            super.processKeyEvent(e);
        }

        if (e.getID() == KeyEvent.KEY_PRESSED &&
            e.getKeyCode() == KindleKeyCodes.VK_FIVE_WAY_SELECT)
        {
            this.dateSchedule.setTimes(specificSchedule);

            return;
        }

        super.processKeyEvent(e);
    }

    // Configures labels to change schedule displayed when cursor selects them
    protected void processFocusEvent(FocusEvent e) {

        this.dateSchedule.setTimes(specificSchedule);

        return;
    }

}
```

****

sheduleItem.java

```
/**
 *
 * @author Sam August
```

```
 */
package schedule;

import com.amazon.kindle.kindlet.ui.KTextArea;

public class scheduleItem extends KTextArea{

    protected boolean toggle = true;

    scheduleItem() {

        // Could not find a way to design a text box to hold more letters than
        // than it had when it was first created.  The work around is to create
        // the text box with lots of spaces so when it is edited it will hold
        // the information.
        this.setText("
");
        this.setSize(600,20);
        this.setEditable(false);
    }

    public void clearText() {

        this.setText(" ");
        this.repaint();
    }
}
```

****

schedule.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package schedule;

/**
 *
 * @author Sam
 */
public class schedule {

    // Make space for each hour of the day
    public scheduleItem sevenAM = new scheduleItem();
    public scheduleItem eightAM = new scheduleItem();
    public scheduleItem nineAM = new scheduleItem();
    public scheduleItem tenAM = new scheduleItem();
    public scheduleItem elevenAM = new scheduleItem();
    public scheduleItem twelvePM = new scheduleItem();
    public scheduleItem onePM = new scheduleItem();
    public scheduleItem twoPM = new scheduleItem();
    public scheduleItem threePM = new scheduleItem();
    public scheduleItem fourPM = new scheduleItem();
    public scheduleItem fivePM = new scheduleItem();
    public scheduleItem sixPM = new scheduleItem();
    public scheduleItem sevenPM = new scheduleItem();
    public scheduleItem eightPM = new scheduleItem();
    public scheduleItem ninePM = new scheduleItem();
    public scheduleItem tenPM = new scheduleItem();

    // Grabs data from input schedule and populates this schedule
    // Note that text areas need to be repainted to refresh on screen
    public void setTimes(schedule input) {
        this.sevenAM.setText(input.sevenAM.getText());
```

```
        this.sevenAM.repaint();
        this.eightAM.setText(input.eightAM.getText());
        this.eightAM.repaint();
        this.nineAM.setText(input.nineAM.getText());
        this.nineAM.repaint();
        this.tenAM.setText(input.tenAM.getText());
        this.tenAM.repaint();
        this.elevenAM.setText(input.elevenAM.getText());
        this.elevenAM.repaint();
        this.twelvePM.setText(input.twelvePM.getText());
        this.twelvePM.repaint();
        this.onePM.setText(input.onePM.getText());
        this.onePM.repaint();
        this.twoPM.setText(input.twoPM.getText());
        this.twoPM.repaint();
        this.threePM.setText(input.threePM.getText());
        this.threePM.repaint();
        this.fourPM.setText(input.fourPM.getText());
        this.fourPM.repaint();
        this.fivePM.setText(input.fivePM.getText());
        this.fivePM.repaint();
        this.sixPM.setText(input.sixPM.getText());
        this.sixPM.repaint();
        this.sevenPM.setText(input.sevenPM.getText());
        this.sevenPM.repaint();
        this.eightPM.setText(input.eightPM.getText());
        this.eightPM.repaint();
        this.ninePM.setText(input.ninePM.getText());
        this.ninePM.repaint();
        this.tenPM.setText(input.tenPM.getText());
        this.tenPM.repaint();
    }

    // Sets specific time values in a schedule.  Used when getting data from
    // external text file
    public void setDate(int index, String words) {

        switch (index)
        {
            case 0: sevenAM.setText(words); break;
            case 1: eightAM.setText(words); break;
            case 2: nineAM.setText(words); break;
            case 3: tenAM.setText(words); break;
            case 4: elevenAM.setText(words); break;
            case 5: twelvePM.setText(words); break;
            case 6: onePM.setText(words); break;
            case 7: twoPM.setText(words); break;
            case 8: threePM.setText(words); break;
            case 9: fourPM.setText(words); break;
            case 10: fivePM.setText(words); break;
            case 11: sixPM.setText(words); break;
            case 12: sevenPM.setText(words); break;
            case 13: eightPM.setText(words); break;
            case 14: ninePM.setText(words); break;
            case 15: tenPM.setText(words); break;
            default: break;
        }
    }

}
```

APPENDIX D: Future Development Guides

i. ACCESSING THE KINDLE 3 ROOT SHELL

At the time of this writing, the Kindle was loaded with Jailbreak v0.6.N and USBNetwork hack v0.32.N. Any updates to these hacks can be found on the MobileRead forums: (http://www.mobileread.com/forums/showthread.php?t=88004). The forum post used to create this guide can be found here: (http://www.mobileread.com/forums/showthread.php?t=117033)

In order to access the Kindle's root filesystem we will take advantage of two hacks developed by members of the MobileRead Forum community: Jailbreak and USBNetwork. The jailbreak, as expected, opens the system for other hacks to be installed. USBNetwork will restore functionality of the Kindle's USB networking mode in which the Kindle acts as a network device when connected via USB. Once it is a network device we can SSH/Telnet to it and gain root access.

Once these hacks have been installed (as they should currently be), we can activate or deactivate the USB networking mode by first entering the debug mode (type ;debugOn in the search bar and hit enter), and then typing ~usbNetwork in the Kindle search bar and hit return. At this point accessing the root depends on your computer's operating system

<u>For Windows 7/Vista Users:</u>

Install Microsoft Windows Mobile Device Center so the Kindle can be recognized as an RNDIS/Ethernet controller. As of this writing the download can be found here:

> *http://download.cnet.com/Microsoft-Windows-Mobile-Device-Center-for-Windows-Vista-32-bit/3000-2094_4-10714488.html*

Under your computer's network (Network and Sharing Center -> Manage network connections) a new Local Area Connection will appear. Right-click that new connection and go to properties. Double-click "Internet Protocol Version 4 (TCP/IPv4)" and set a specific IP Address of 192.168.0.1, subnet mask 255.255.255.0. Click okay on both open windows and return to the listing of your Network Connections.

Now open the properties for the Local Area Connection through which your computer is connected. There should be a "Sharing" tab available in the window. Go to it and check "Allow other network users to connect through this computer's internet connection". Hit okay, and a dialog should appear confirming that this action will change the adapter interface to 192.168.0.1, which you should agree to. Once again go back to the new Local Area Connection's properties and change the IP address to

192.168.2.1. Finally, open a command prompt and enter “telnet 192.168.2.2”. Now you should be connected to the Kindle to access all the internal files.

ii. CREATING JAVA APPLICATIONS FOR THE KINDLE

This guide was written specifically for Windows based computers, however the process is still necessary to make a Java application run on the Kindle. The forum post used to create this guide can be found here: (http://www.mobileread.com/forums/showthread.php?t=102386)

Creating applications for the Kindle requires that the .jar files created by the Java IDE are signed by keystores found on the Kindle. Since we lack the signature information from Amazon, we need to create a custom keystore to sign our applications and load it onto the kindle. To do this we will use the keytool application included in the Java Runtime Environment folder. Using this tool involves opening a command prompt and navigating to the Java runtime environment's binary folder (In my case it was located in /Program Files/Java/jre6/bin). Once there, three commands need to be run:

keytool –genkeypair –keystore developer.keystore –storepass YOUR_PASS_HERE –keypass YOUR_KEY_PASS_HERE –alias dnYOUR_ALIAS –dname "CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown" –validity 5300

keytool –genkeypair –keystore developer.keystore –storepass YOUR_PASS_HERE –keypass YOUR_KEY_PASS_HERE –alias diYOUR_ALIAS –dname "CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown" –validity 5300

keytool –genkeypair –keystore developer.keystore –storepass YOUR_PASS_HERE –keypass YOUR_KEY_PASS_HERE –alias dkYOUR_ALIAS –dname "CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown" –validity 5300

This will create a keystore that can be used to sign applications. An example of the code used to create the keystore that was used to sign the first version of the schedule application is below for reference:

keytool –genkeypair –keystore developer.keystore –storepass password –keypass password –alias dkTest –dname "CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown" –validity 5300

As of this writing, the keystore used to sign applications is developer.keystore, using the alias "Test" and password "password". Once the keystore has been created, it needs to be loaded onto the Kindle. Using either the USB networking mode or another method, access the /var/local/java/keystore folder on the Kindle's internal filesystem, and copy the custom keystore into that folder. Once the keystore is placed, java applications are ready to be signed and run on the Kindle.

Signing the applications works much like creating the keystores, but a little simpler. Once the .jar is created, open a command prompt and navigate to the Java Development Kit binary folder (in my case, it was /Program

Files/Java/jdk1.6.0_16/bin). Once in that folder, three commands need to be run again:

```
jarsigner –keystore KEYSTORE_FILE –storepass KEYSTORE_PASS JAR_FILE
    dkYOUR_ALIAS

jarsigner –keystore KEYSTORE_FILE –storepass KEYSTORE_PASS JAR_FILE
    diYOUR_ALIAS

jarsigner –keystore KEYSTORE_FILE –storepass KEYSTORE_PASS JAR_FILE
    dnYOUR_ALIAS
```

An example of the code used to sign the first version of the application is shown below for reference:

```
jarsigner –keystore /Users/Sam/Desktop/developer.keystore  –storepass password
    /Users/Sam/Desktop/Schedule.jar dnTest
```

After the jar has been signed, the file extension needs to be changed from .jar to .azw2. To do this simply rename the file (with file extensions showing) and accept the dialog warning that the file may become unstable. The programs on the Kindle are in essence .jar files, but the Kindle reads a different file type so the change needs to happen. Now that the file is of the correct type, the application can be loaded onto the Kindle into the documents folder, and can be opened from the home menu.

iii. UPLOADING A SCHEDULE

Editing the schedule that is displayed on screen is a very simple process. Plug the kindle into a personal computer using the provided USB cable. An auto-play dialog should appear, and you should select “Open folder to view files”. From here navigate to the folder /developer/Johns Schedule/work, and in this folder you will find a file titled “schedule.txt”. Please note that if the title of the application changes, the “Johns Schedule” portion of the filepath will also change. Open the schedule.txt file. You will see a listing of each day of the week, with hours of the day labeled below. The days of the week are used purely as labels for the user to make the editing process more clear. As a result, the program will skip those lines during data acquisition. If you choose to remove or these labels, please leave an empty line in their place, or the program will not operate as intended. When acquiring data the program takes the entirety of the line for the hours of the day and displays it, so time labels can be changed as needed. Again, like the day labels, if you choose to remove hours of the day, please leave a blank line in their place for correct program operation. Once the editing is complete, the text file can be saved and the Kindle can be unplugged from the computer. If the application was open it must be restarted in order for any changes to the schedule to take effect.