GREEN PHONES PHASE 1: PROJECT REPORT

Adrian Kwok (#200136359)
Zachary Blair (#200106230)
Benjamin Saunders (#301111447)

CMPT 885
Professor Shriraman
August 22nd, 2011

1. INTRODUCTION

For our semester-long project for CMPT885, we contributed to Professor Shriraman's and Professor Fraser's Green Phones project, developing the underlying framework for which future work can be built upon. The Green Phones project aims to understand and concretely define factors that can influence overall power consumption in a modern Android-based smartphone, with the ultimate goal of being able to provide strong power consumption estimates for *any* given application available on the Android Market. Overall, the project requires work on several different components to formulate a successful, publishable paper. At its bare minimum, it is necessary complete the following phases:

1. Gather sufficient power consumption measurements while varying different resource loads to develop an accurate power estimation model. This requires simulating many, many different inputs (resource loads) and generating corresponding outputs (accurate total power consumption of the phone). This was the entire focus of our work for the project;
2. Verify the correctness and quantify the total amount of error in the generated power estimation model;
3. Given an Android application (e.g., from the Android Market[1]), be able to determine what resources it uses, and how much, at a fine-grained level;
4. Utilize the application's measured resources as an input to the developed model to provide an accurate estimation of the application's power consumption;
5. Compare different applications that accomplish the same task and see which are the most power efficient.

As a result of working on this project, a number of future possible research avenues were also discussed:

a. Determining and comparing the effectiveness of different dynamic frequency scaling algorithms with respect to minimizing CPU power consumption;
b. Concretely defining the benefits and drawbacks to the Android OS's laissez-faire attitude with resourced-based wake locks[2][3], especially with the proliferation of poorly written applications available on the Android Market[4]. On the issue of user control, Wysocki[3] writes:
   "Of course, processes using wakelocks can impact the system's battery life quite significantly, so the ability to use them has to be regarded as a privilege that should not be given unwittingly to all applications. Unfortunately, however, there is no general principle the system designer can rely on to figure out what applications will be important enough to the system user to allow them to use wakelocks by default. Therefore, ultimately the decision is left to the user which, naturally, is only going to really work if the user is qualified enough to make the decision."
c. Compare the power consumption of a specific component over a variety of different smartphones when running the same application, to find out energy efficiency of newer or faster components.

While there is still a considerable amount of work to be done, we feel that we have made a significant contribution to the project – we successfully and *carefully* developed fine-grained user-level component stressors, managed to interface with a phone's battery to gather accurate power measurements, and essentially assembled all the necessary tools and test beds required to begin work on the next phase of the project. listed above as phase 2. The amount of possible research that can be conducted on this topic is vast, and considering the current paradigm shift to widespread handheld computing coupled with the physical limitations of batteries, such research should be extremely relevant and important in the near and distant future.

2. STIMULATING RESOURCE LOADS (STRESSAPP)

To simulate a variety of different resource loads on most of the components on a conventional smartphone, we built a user-level stress application – aptly named "StressApp"– on the Android platform. This application allows researchers to specify time intervals for which each component is to be under a user-defined amount of stress, with minimal additional overhead from component coordination. Moreover, StressApp synchronizes with the hardware power logger – discussed in section 3. – for each stress run, simplifying the data gathering process used to build a power estimation model. Coupled together with the ability to easily create fine-grained state changes in a stress run, the power estimation model can be built very carefully and accurately.

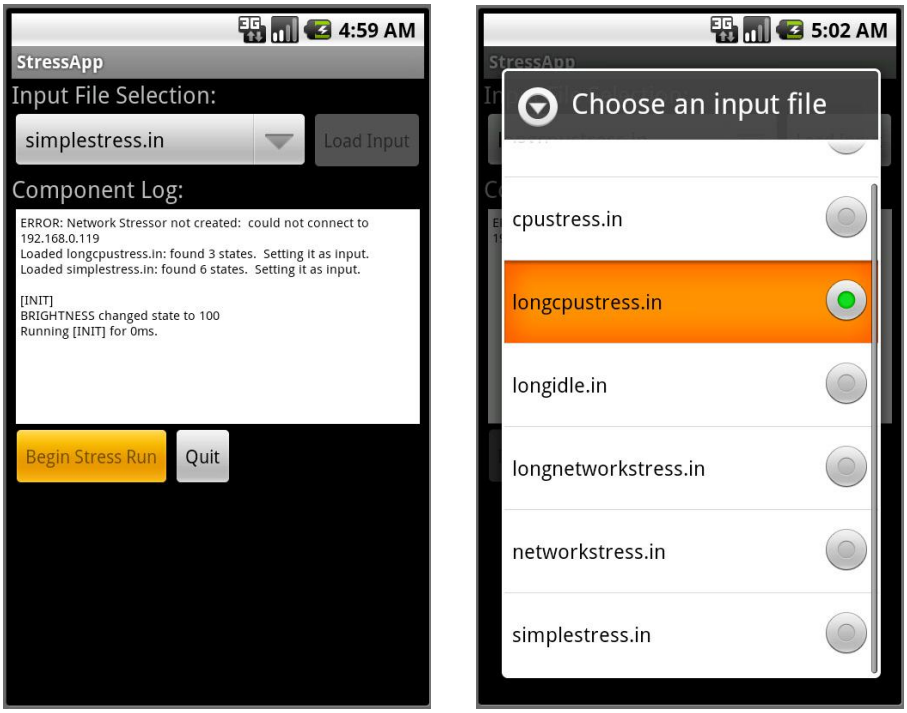

Figure 1: The StressApp GUI

The overall StressApp architecture comprises of a several major modules:

1. **An input parser.** The input parser reads in component stress run configurations and extracts states comprising of requested component loads and state durations. This is further elaborated upon in Section 2.1.

2. **The application GUI thread/activity.** The GUI simply allows the user to choose from a variety of stress run configurations stored in the "stress" folder of the SD card and execute them, while showing stress run progress and details. The GUI does not interact with any of the component stressors

3. **Component stressor threads.** In our application, each component is a separate thread, allowing for multiple components to be stressed simultaneously, analogous to real-world smartphone usage. The general stressor framework, discussed in Section 2.2, allows specific component stressors to *easily* and *efficiently* switch between different stress states. Specific component stressor implementation details are discussed in Section 2.3.

4. **The master thread.** The master thread coordinates all of the component stressor threads: it creates each stressor thread in its `createStressors()` method and places them into a `HashMap` data structure, where each entry's key is a user-defined `enum` from Components.java and the value its corresponding stressor thread. This was purposely designed after feedback from Professor Fraser: if a new component stressor is created, incorporating it into StressApp requires only adding two lines of code in `createStressors()`, and a corresponding `enum` in Component.java. The master thread only acts upon entries in the HashMap.

   After creating all the component stressor threads, the master thread sleeps and waits for a stress run to begin. When it is told to start a stress run by the GUI, it is interrupted from its sleep and, for each state in the run, tells all stressors to change state, and then sleeps for the duration of the state – effectively running each component at the desired load for that amount of time – before moving on to the next state. Immediately after the init and immediately before the end states, the master thread synchronizes with the hardware power logger described in Section 3 via a few HTTP GET calls. When a run is completed, it goes back to sleep.

2.1 Input File Format & Parser

Test configurations are represented in a simple file format designed for easy extensibility and manual editing. This was achieved by modeling after the common INI-style structured configuration format, which allows for sufficient expressiveness for our purposes while permitting a straightforward approach to parsing without compromising readability. Attempts were made to take advantage of existing parser code sourced from third parties to implement this design, but initial research and experiments turned up no libraries both lightweight enough to be easily included in the application and supportive of particular expressive needs of the format such as retention of section ordering, and so a parser was manually implemented.

A test file consists of 2 or more sections each containing zero or more key-value pairs, and any number of comments. A section is designated by a line beginning, disregarding whitespace, with "[" and containing a subsequent "]", all other contents of the line being ignored; in practice, this is a good place to put a one-word title or description of the section. Key-value pairs consist of a line which is not a section designator or a comment and which contains an "=" character. All text before the "=", minus leading and trailing whitespace, is considered to be the key, and all text after, similarly trimmed, is considered to be the value. If multiple "=" are present on a line, the first occurrence is used. Note that key-values may exist outside of a section by being placed before any section designator. Such key-values are ignored. Finally, comments are lines whose first non-whitespace character is a ";" or a "#"; these lines are also ignored. Note that comments are only supported on otherwise empty lines.

```
[init]
brightness=100
time=0

[S0]
time=10000
brightness=10
cpu=10
audio=10

[S1]
time=10000
brightness=20
cpu=20
audio=20

[end]
audio=off
brightness=100
time=0
```

Example Input File

The first and last sections have special meaning: the first is applied prior to the beginning of data logging (but after a test run execution is requested), and the last is applied immediately after data logging terminates. These are intended to be used to ensure that a test begins from a reproducible state, potentially including changes such as full-screen image display which might make user interaction with the stressor program hard, and returns the system to a sane state upon completion. All other states are executed in the sequence they occur in the file, potentially being repeated some number of times before the end state is executed and control is returned to the user.

Key-value pairs are used to transition individual stressors into a particular state at a given stage. Keys correspond to a particular stressor, and values to the state desired for that stressor to enter into when the current section is executed. As the full set of stressors needed cannot be predicted with certainty at this time, it was decided to define values as arbitrary (one-line) strings, on the basis that this permits arbitrary data to be easily passed to stressors as needed without requiring complex parser and configuration infrastructure. Keys, however, are more constrained: each section may contain zero or one key-value pair for each entry in the Component `enum`. For sections other than the first and last, this must include a "time" special (in that it does not correspond to a stressor) key, whose value indicates the amount of time for which the states should be sustained before executing the next section. The configuration loader is implemented in terms of this `enum`, so the addition of further stressors should require no further changes other than the addition of a new element to this `enum`, and, of course, the use of the corresponding key in some stressors. Interacting with the configuration loader itself is straightforward: construct it on a file containing a configuration, and then query initial, final, and normal states for arbitrary Components using `getInitState()`, `getEndState()`, and `getState()` respectively, with normal state indexes ranging from zero to one less than `getNumStates()`.

2.2 Stressor Framework

Stressors themselves are subclasses of the Stressor class, which must override the `applyState()` and `stress()` methods. The `applyState()` method is called on a string obtained from the value of a key-value pair from a test configuration input file, and it must configure the stressor such that the next call to stress will invoke the desired behavior. Note that the string is visible only to `applyState()`, and may safely be modified without side-effecting other code. `applyState()` must not perform any actual stressing itself, as calls to it may occur when action is not actually desired. However, it should perform any necessary parsing of the configuration string. The `stress()` method must take the action implied by a given state, such as CPU loading or transfer of data over the network. It must also either check the interrupt status of the thread by inspecting the return value of `interrupted()` frequently and throwing an `InterruptedException` when it is true, or by performing calls (such as sleep or certain I/O methods) which are known to themselves throw `InterruptedException`s when their thread is interrupted. This returns control to the superclass, which manages communication between the stressor thread and the master thread before resuming stressing if appropriate. This approach was used to help simplify stressor implementation by allowing blocking I/O and sleeping to be used freely, as thread interruption tends to abort these operations.

The master thread interacts with the stressor threads through the `begin()`, `end()`, `running()`, and `changeState()` methods. The first two cause calls to `stress()` to begin and cease occurring, respectively, while `running()` simply indicates whether stress is currently being (or soon to be) called, and `changeState()` causes `applyState()` to be invoked on a copy of the supplied string in the stressor thread. Note that it is possible for `changeState()` to be called more rapidly than the child thread can invoke `applyState()`, potentially resulting in intermediate states being skipped. However, if stressors check the interruption flag with sufficient frequency this should only occur when the time interval between two calls to `changeState()` is sufficiently small that the skip makes little measurable difference. Also note that, as all overrideable methods are executed in the stressor thread – no locks or synchronization beyond that already implemented should be necessary.

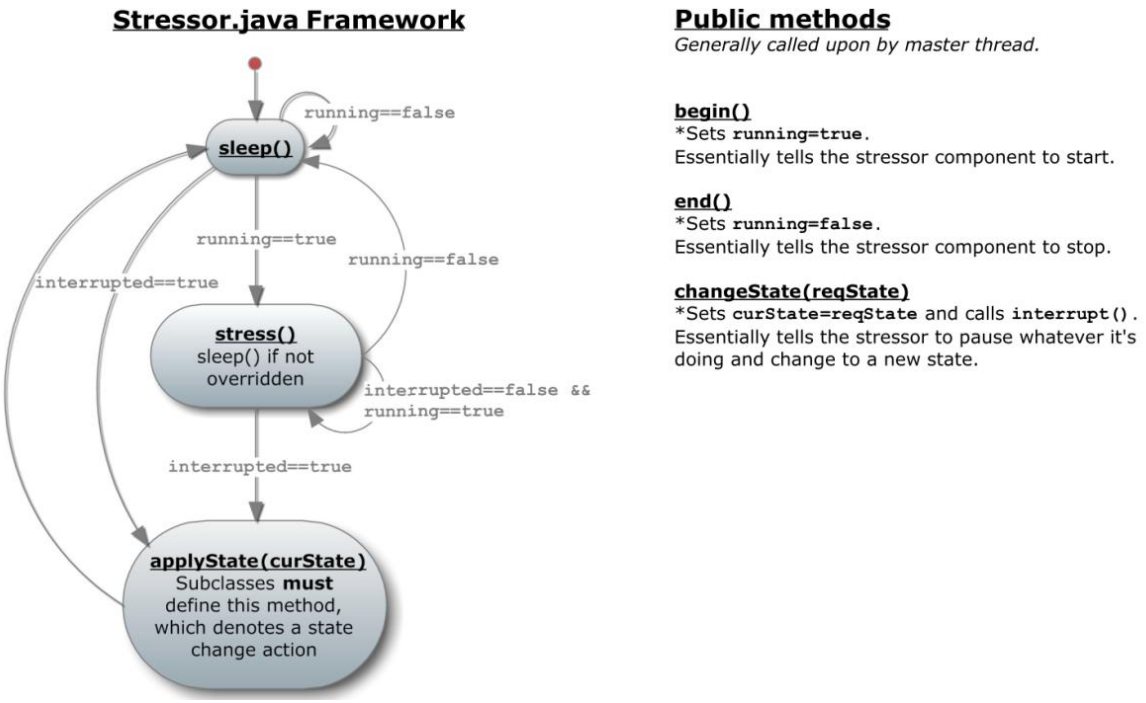

Figure 2: State Diagram for Stressor Framework

This design allows individual stressor implementations to take full advantage of the generality of the test configuration format and minimizes the amount of complexity necessary for new additions, all while maintaining good performance and helping maintain the readability and simplicity, and thus long-term maintainability, of the remainder of the StressApp implementation.

2.3.1 CPU Stressor

For the CPU stressor, we utilized a Linpack Java port from initially written in C[4]: Linpack based benchmarks, such as LinpackX and IntelBurnTest, are applications widely used to stress PCs to verify overclocking stability. Moreover, Linpack is more indicative of a true, heavy CPU workload compared to just repeatedly adding a constant to a variable as done in PowerTutor[5]. With regards to previous discussion on the general stressor framework, the CPU stressor simply keeps running a Linpack benchmark in the `stress()` method, and changes the target CPU utilization variable in the `applyState()` method.

Since it is necessary to be able to vary the CPU utilization in developing a fine-grained power estimation model, we spliced "checkpoints" into different areas of the Linpack Java code; these checkpoints – defined by the `checkThrottle()` method – check to see if a predetermined time has elapsed since the previous checkpoint, and if so, tells the thread to sleep to reach an average target CPU utilization level. As a contrived example, a checkpoint can check if 1 second has elapsed, and if so, sleep for 1 second to achieve an average CPU utilization of 50% over 2 seconds. More generally, the relationship between elapsed time, sleep time, and target CPU utilization is given by:

$$\frac{elapsed\_time}{elapsed\_time + sleep\_time} = cpu\_util$$

where $0 < cpu\_util \leq 100$. However, since we wish to develop a fine-grained power estimation model, with an ideal polling interval of 200Hz, i.e. 5ms, some difficulties arise. Depending on the CPU the stressor is being run on, the time between consecutive checkpoints may be so large that it is unable to reach a target CPU utilization accurately – more specifically, if we aim for a 99% CPU utilization over 5ms, this would require us to monitor for a period of 0.05ms (and sleeping for 4.95ms). On the other hand, if we aim for a 1% CPU utilization over 5ms, this would require us to sleep for 0.05ms *consistently*, which seems to be a difficult task for most operating systems[7][8][9].

To alleviate this issue, we require that the target CPU utilization be defined in steps of 10%, resulting in an absolute minimum elapsed time of 0.5ms when the target CPU utilization is 10%, and the same time for sleeps when the target CPU utilization is 90%. Since the minimum time interval is 0.5ms, we can simplify the formula discussed previously, and the actual elapsed and sleep times used in the stressor is given by:

$$if\ (cpu\_util > 0.5): \begin{cases} elapsed\_time = \frac{cpu\_util}{1-cpu\_util} \cdot 0.5ms \\ sleep\_time = 0.5ms \end{cases}, \quad else: \begin{cases} elapsed\_time = 0.5ms \\ sleep\_time = \frac{1-cpu\_util}{cpu\_util} \cdot 0.5ms \end{cases}$$

As a side note, there was some initial trouble with getting the stressor thread to sleep for values of less than 1 millisecond – necessary for a 5ms polling interval. While the generic `Thread.sleep()` method allows for nanosecond input, it seemed to round up the nearest millisecond – to solve this, we instead relied on the `parkNanos()` method offered by `LockSupport` in Java's `util.concurrent` package[10], which seems to work as expected[11].

2.3.2 Memory Stressor

For the memory stressor, we simply allocated a large array of 8-byte integers, and wrote to the array based on two interaction types: sequential or random access. In both cases, the `applyState()` method simply sets the stressor to one of these types, and the `stress()` method interacts with the array based on this – careful attention was made to ensure minimal wasted CPU cycles (e.g., we only check for the access type *once* in each stress run, as opposed to at every single element access). Furthermore, since array allocation is costly, we only allocate the array once when the stressor is constructed, outside of any timing run.

Implementing sequential writes is trivial: it was implemented by simply writing through the entire array in a single loop, while checking the thread's interruption status at every element access. In the `stress()` method, this corresponds to running the this loop infinitely many times.

On the other hand. random writes are a bit more tricky: the naive implementation would simply run a random number generator at each access request and write to that random element – however, there is considerable CPU overhead with generating random numbers, and we do not want it to influence our timed results. Instead, we can create a single lookup table – created prior to all timing runs – and allocated to be the same size as the array being accessed where each element in the lookup table corresponds to an index to write to in the main array. For example, if the array we write to has 5 elements, then the lookup table could look like the following:

4	2	0	1	3

This example lookup table would correspond to a write to index 4 of the main array, followed by a write to index 2 of the main array, a write to index 0 of the main array, and so forth. A lookup table is preserved

through the entire lifetime of the stressor, and the `stress()` method simply iterates through the lookup table one element at a time, writing to the corresponding element in the main array, and checking the thread's interruption status. Generation of the lookup table is done in the stressor's constructor, done by shuffling an array with incrementing elements.

Note: while we have not formally implemented an SD card/storage stressor, it is simple to port the memory stressor code over, as it is almost identical to what is necessary for those stressors.

2.3.3 Network Stressor

For the network stressor, we established two TCP connections to a remote server – usually tested under the same LAN, and in many cases running on the same machine as the PowerLogger – and sent or received multiple 1KB blocks per second over these connections. According to PowerTutor[12], the main determinant of power consumption in WiFi are the number of packets sent per second, allowing the network adapter to switch between low, medium, and high-power states – as such, we allow the user to request different bandwidth (KB/sec) amounts on a state change.

For the remote server, we ultimately settled on using two simple `netcat` commands to receive and send data: "`nc -l 1234 > /dev/null`", which listens on port 1234 and, on receipt of data, forwards it to `/dev/null`, effectively trashing it, and "`nc -l 1235 < /dev/urandom`", which listens to port 1235, and, on request, sends garbage randomized data efficiently. Unfortunately, for GNU-based variants of `netcat`, a keep-alive function is not supported – meaning that once a client disconnects from the server, `netcat` stops running – so we utilized simple bash scripts to loop the `netcat` command until it is manually terminated, allowing for multiple network stressor runs without issue. These scripts are located in SVN under the filenames "server_send.sh" and "server_receive.sh".

When the network stressor (client) is created, it opens two sockets and handshakes via TCP with the remote server; this is done so that the additional overhead doesn't affect the results of a timed run. On the network stressor's `applyState()` function, we specify a requested bandwidth rate and network type (either download or upload), and in the `stressor()` method we send or receive 1KB blocks using the two sockets created previously. To make sure that we adhere to the bandwidth rate requested, we sleep for an amount of time after each send/receive proportional to it – e.g.., if we desire a steady 100KBps rate, we would sleep for some amount around 10ms per send/receive (not exactly 10ms since the act of sending/receiving the block takes time as well) – and, if we end up sending too many blocks over a 1 second period, we sleep for an amount of time that would put us back on track.

Note: the network adapter used, i.e. WiFi versus EDGE/3G, is determined by the phone's setting prior at the time of a stressor's run. At this point in time, functionality to switch between adapters on `applyState()` has not been implemented yet, but this is a simple addition as turning off the WiFi (which would, in most cases, default the adapter to 3G) is fully supported through the SDK. Unfortunately, turning on or off 3G manually is not something that is offered by the SDK.

2.3.4 Audio Stressor

For the audio stressor, we loop a small 150KB, 2-channel, 44.1Khz sampling rate uncompressed white noise audio file, at varying requested volume levels. As we wish to minimize CPU overhead, we utilize Android's simpler `SoundPool` library to play this file as opposed to the `MediaPlayer` library, which allows for much more functionality but is overkill for our purposes. When the audio stressor is first created, we initialise and load the audio file into a `SoundPool`. On an `applyState()`, we store the desired volume such that the stressor can change the volume when `stress()` is next run – recall that an `applyState()` could occur a long time before `stress()` is called, if we're just setting initial states prior to a `begin()` call. In the `stress()` function, we do the either the following *once*: the stressor either plays the audio file via `SoundPool` at the desired volume, or if it's already playing, it simply changes the volume. Afterwards, the stressor just sleeps.

2.3.5 Brightness Stressor

Similar to the audio stressor, in the brightness stressor we simply record the desired brightness in an `applyState()`, and set the brightness *once* when `stress()` is called, otherwise `stress()` just sleeps.

There were two problems observed while implementing this stressor:

- If the desired brightness is 0% (i.e. screen is off), the phone immediately goes to sleep and the whole application stops. We used a `PARTIAL_WAKE_LOCK`[2] to try to remedy this problem, and while the application continues to run and the screen is turned off, it is not possible to change the brightness again afterwards. This problem has been documented online in numerous forums[13][14][15][16], and after much searching an adequate solution still has not been found. As it stands, if a stress run wishes to turn off the screen, it will have to do so for the remaining duration of the run.
- Only the main activity thread (i.e. UI/master thread) is able to change the brightness, and no other thread is able to, otherwise an error is thrown. We remedied this by using a Handler to allow the stressor thread to send a message to the main activity thread to set the actual brightness.

3. Logging Power Consumption (PowerLogger)

We developed an electrical power consumption measurement and logging infrastructure that enables researchers to physically measure and log power consumption of the smartphone during various tests. The logging infrastructure consists of three components: the battery gas gauge instrumentation board, the interface board, and a logger application for the PC. The figure below shows the general configuration of the logging infrastructure.

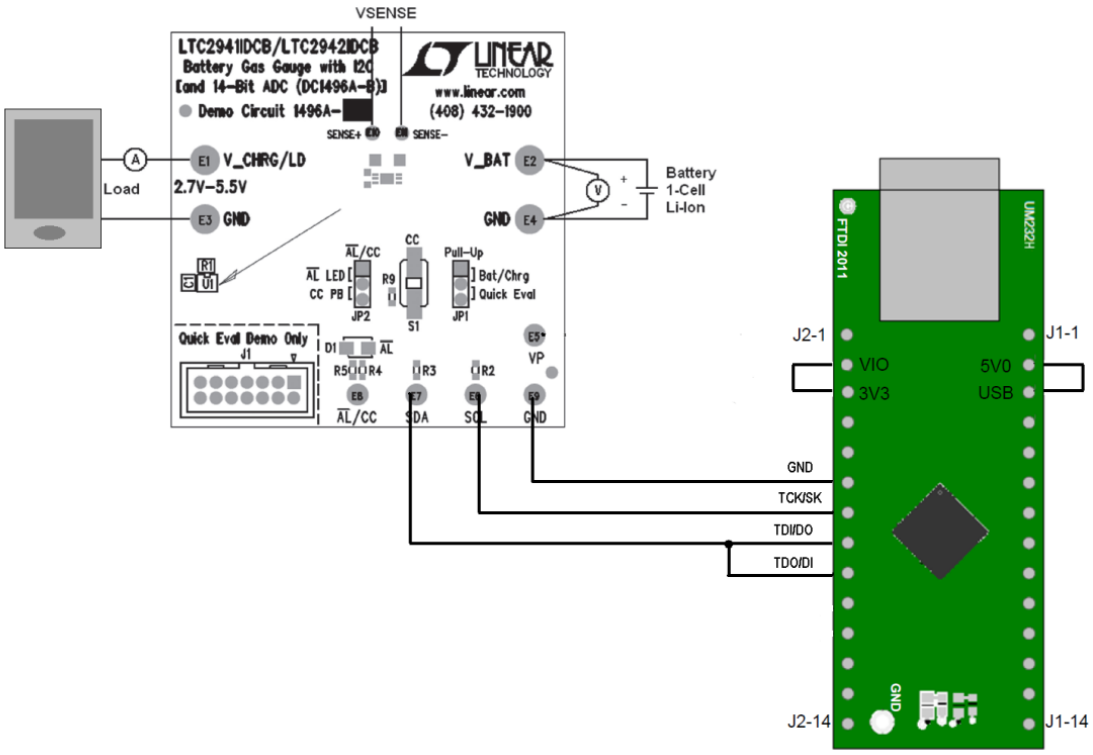

Figure 3: Power Logging Hardware Configuration

The battery from the smartphone has been removed from its chassis, and attached to the smartphone via some wires that first pass through the battery gas gauge board. This was done so that the battery gas gauge board can actually measure the battery voltage and current flowing into the smartphone. Using this voltage and current information, the board is able to accurately estimate the amount of charge, measured in Coulombs, that has flowed out of the battery into the smart phone, thereby providing an accurate estimate of the remaining battery capacity.

3.1 Battery Gas Gauge Board

The battery gas gauge board, a Linear Technology 1496 Demo Board, is based on a LTC2942IDCB chip, described by the Linear Technology data sheet as a "Battery Gas Gauge with I2C and 14-Bit ADC". The board measures the battery current and voltage, and uses it to estimate the battery's power output[17]. By reading the values of certain registers on the board via its I2C interface, our logger application can monitor the phone's power consumption with high accuracy.

The board has an I2C/SMBus interface for communication with another board. The I2C/SMBus interface is a two-wire system, with all devices connected in parallel to the wires, and both wires pulled-up with pull-up resistors. The two wires, SDA and SCL, provide a simple interface involving a data line, and a clock line:

I2C Line	Purpose
SDA (Serial Data Line)	Used to transmit data. The wire is weakly pulled up by a pull-up resistor, so that any connected device can pull it down to transmit.
SCL (Serial Clock)	Controlled by the "Master" I2C device. Used to synchronize communication.

In addition to the I2C interface, the LTC 1496 Demo Board also provides a special pin for notifying external board of events, such as when charging of a battery is complete.

Pin	Purpose
~AL/CC (Alert Output or Charge Complete Input)	Used to notify other chips of certain events, such as when the battery has finished charging.

We didn't end up using that pin, and instead rely on the I2C interface exclusively.

3.2 I2C to USB Convertor Board

The battery gas gauge board only has an I2C/SMBus two-wire interface, which our PC cannot directly interface with. Therefore, we used a FTDI *UM232H* Single Channel USB Hi-Speed FT232H Development Module board to provide an interface between the battery gas gauge board's I2C interface and the PC's USB (Universal Serial Bus) port.

The FTDI website provides several drivers for Windows, Mac, or Linux for interfacing with their UM232H USB devices[18]. One set of drivers makes the device look like a normal serial "COM:" port, and another allows for more fine-grained control using a proprietary API. The logger application uses the D2XX drivers available from the FTDI website to acquire data from the LTC2941 board.

3.3 PC Logger Application

A major component of the logger infrastructure is an application running on a connected PC, which continually communicates with the battery gas gauge board, and logs measured voltage, current, and battery levels to a file.

We used the Qt framework[19] for the logger application's GUI, because it is cross-platform (Windows, Mac, Linux), extensive, and also because of our familiarity with it. We created a logger GUI application, and have committed it to the SVN repository under the "/PowerLogger" directory. The following is a screenshot of what the program looks like on Windows XP.

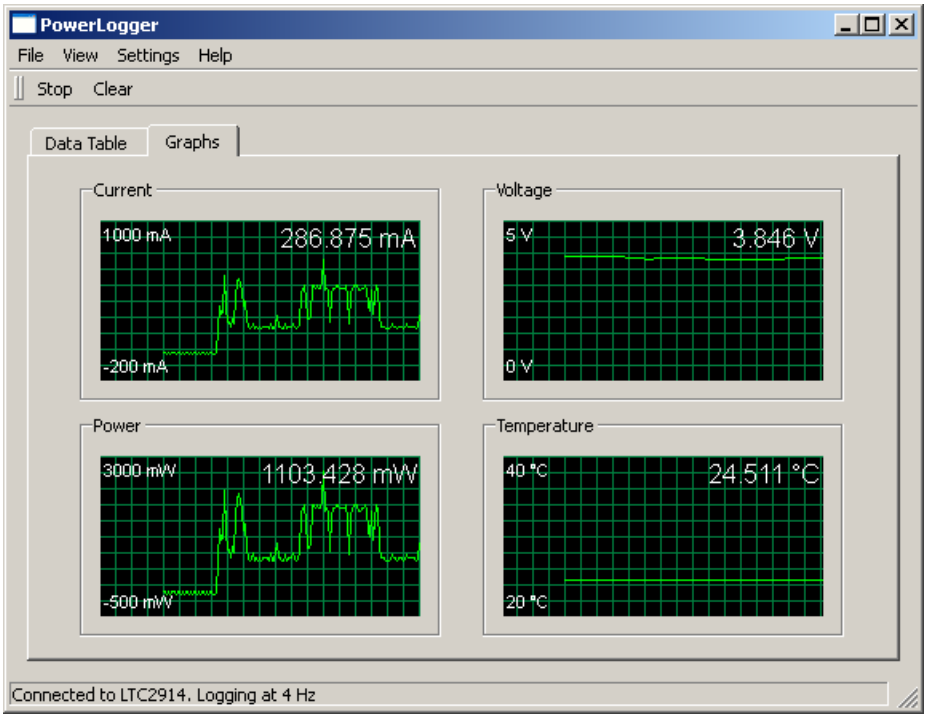

Figure 4: PowerLogger GUI (Graphs View)

The application provides a simple GUI for starting and stopping logging, and for clearing any previously logged data. It also provides an interface for visualizing the data in a graph, or in a spreadsheet. In the screenshot above, the "Graphs" tab shows the data in a graph, and the "Data Table" shows the 10 most recent measurements in a table.

The battery gas gauge board only provides charge counter information, voltage information, and temperature information. Therefore, the "Current" and "Power" data shown in the graphs had to be derived from consecutive charge and voltage measurements.

Current is simply the rate of change of the charge counter, so it was derived using the following formula:

$$i_n = \frac{q_n - q_{n-1}}{t_n - t_{n-1}}$$

where q_n and q_{n-1} denote the nth and n-1th charge counter values, respectively, and t_n and t_{n-1} represent the nth and n-1th timestamp values, respectively.

The "Power" graph is then derived from the current and voltage information using the following formula:

$$p_n = i_n \times v_n$$

where v_n is the nth voltage measurement.

3.4 PC Logger Application: Remote Control Interface

The logger application also provides an HTTP interface for remotely starting, stopping, clearing, and saving the log. This interface may be accessed using a standard web browser application like Firefox, Chrome, or Internet Explorer, or by a custom application. The testing application on the Android phone, for instance, connects to the logger application on the PC using this HTTP interface so that it can start and stop the logger in synchronization with the tests.

The table below lists all the URLs that the logger application exposes. Sending an HTTP GET request for these URLs will perform the corresponding function specified in the "Description" column of the table. Thus, entering these URLs in the address bar of a web browser and pressing enter will cause the logger application to perform the function corresponding to the URL entered. The logger application will return a short confirmation string, such as "Logging started", to the web browser in response.

URL	Description
http://localhost/start?freq=NN	The 'start' URL starts the logger. An optional "freq" parameter can be passed in the URL to specify the frequency at which to poll the battery gas gauge board. If no frequency is specified, the frequency currently configured in the logger application is used.
http://localhost/stop	The 'stop' URL stops the logger.
http://localhost/clear	The 'clear' URL clears the logged data, therby erasing the contents of the data table and graphs in the GUI application.
http://localhost/save?name=log.txt	The 'save' URL saves the currently-logged data to a text file as a table of comma-separated values. The optional "name" parameter is used to specify a file name. If no "name" parameter is specified, a file chooser dialog will prompt the user on the PC for a file name. The saved file can be opened as a CSV file using Microsoft Excel or OpenOffice Calc for further analysis.

Open Source Software Contribution

We could not find a suitable graph widget for the PC logger application, so we wrote our own graph widget, and published the source code online so that other developers could use that widget for their projects. Additionally, by providing the source online, any improvements that other users make to the code can be incorporated into future versions of the PC logger application. The widget is available at [20].

4. REFERENCES

1. *Google, Inc., "Apps – Android Market"*. Retrieved August 20th, 2011. Available at: https://market.android.com/?hl=en
2. *Google, Inc., "PowerManager | Android Developers"*. Retrieved August 20th, 2011. Available at: http://developer.android.com/reference/android/os/PowerManager.html
3. Wysocki, R. J., *"An Alternative to Suspend Blockers"*, November 24th, 2010. Available at: http://lwn.net/Articles/416690/
4. Webster, S., *"Poor education to blame for Android returns, not poor apps"*, June 3rd, 2011. Available at: http://reviews.cnet.com/8301-19736_7-20068744-251.html
5. Dongarra, J., Wade, R., McMahan, P., *"Linpack Benchmark – Java Version"*. Retrieved August 20th, 2011. Available at: http://www.netlib.org/benchmark/linpackjava/
6. PowerTutor, *"A Power Monitor for Android-Based Mobile Platforms"*. Retrieved August 20th, 2011. Available at: http://ziyang.eecs.umich.edu/projects/powertutor/powertutorplus.html
7. StackOverflow, *"Sleep Less Than one Millisecond",* originally posted by "smink", September 17th, 2008. Available at: http://stackoverflow.com/questions/85122/sleep-less-than-one-millisecond
8. Defective Compass, *"High Precision Sleep"*, originally posted on September 1st, 2006. Available at: http://defectivecompass.wordpress.com/2006/09/01/high-precision-sleep/
9. StackOverflow, *"How accurate is python's time.sleep()?"*, originally posted by "Claudiu", July 15, 2009. Available at: http://stackoverflow.com/questions/1133857/how-accurate-is-pythons-time-sleep
10. Oracle Corporation, *"LockSupport (Java Platform SE 6)"*. Retrieved August 20th, 2011. Available at: http://download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/LockSupport.html
11. Hay in a Needlestack, *"High Resolution Timer in Java 5"*, originally posted by Gustavo M. D. Vieira, August 28th, 2007. Available at: http://www.sagui.org/~gustavo/blog/code
12. Zhang, L., et. al., *"Accurate Online Power Estimation and Automatic Battery Behaviour Based Power Model Generation for Smartphones", CODES+ISSS '10, October 24-29, 2010.*
13. StackOverflow, *"Blank screen when restoring screenBrightness Android"*, originally posted by "thegreyspot", July 21st, 2011. Available at: http://stackoverflow.com/questions/6783159/blank-screen-when-restoring-screenbrightness-android
14. StackOverflow, *"Calling hidden API in android to turn screen off"*, originally posted by "David Shellabarger", December 9th, 2009. Available at: http://stackoverflow.com/questions/1875669/calling-hidden-api-in-android-to-turn-screen-off
15. StackOverflow, *"turn the screen on/off in Android with a shake"*, originally posted by "The WebMacheter", March 6th, 2011. Available at: http://stackoverflow.com/questions/5214033/turn-the-screen-on-off-in-android-with-a-shake
16. StackOverflow, *"Android dim screen down problem"*, originally posted by "Kneed", June 8th, 2011. Available at: http://stackoverflow.com/questions/6283555/android-dim-screen-down-problem
17. Linear Technology, *"LTC2942 – Battery Gas Gauge with Temperature, Voltage Measurement"*. Retrieved on August 20th, 2011. Available at: http://www.linear.com/product/LTC2942
18. FTDI Chip, *"Drivers"*. Retrieved on August 20th, 2011. Available at: http://www.ftdichip.com/FTDrivers.htm
19. Nokia Corporation, *"Qt – A cross-platform application and UI framework"*. Retrieved August 20th, 2011. Available at: http://qt.nokia.com/products/
20. Blair, Z., *"zblair/QSimpleTickerGraph"*. Retrieved August 20th, 2011. Available at: https://github.com/zblair/QSimpleTickerGraph